# UNIVERSIDAD AUTONOMA DE MADRID

## ESCUELA POLITECNICA SUPERIOR




# TRABAJO FIN DE GRADO

## SISTEMAS DE RECOMENDACIÓN DISTRIBUIDOS


**Alejandro Castillo Saco**

**Tutor: Saúl Vargas Sandoval**

**Ponente: Pablo Castells Azpilicueta**


**SEPTIEMBRE 2014**

# Resumen

Los sistemas de archivos distribuidos y los sistemas de computación distribuida están ganando una gran importancia en la actualidad debido al aumento en la cantidad de datos que organizaciones y empresas son capaces de obtener y analizar.

Del mismo modo los sistemas de recomendación están más presentes en más diferentes ámbitos tratando de mejorar la experiencia del usuario aumentando su satisfacción.

Este trabajo trata de aunar estos dos campos de la informática, implementando cuatro algoritmos de recomendación sobre dos modelos de computación distribuida: MapReduce y Pregel.

Tras demostrar la posibilidad de implementar dichos sistemas de recomendación sobre ellos explicando cómo se ha llevado a cabo, se experimentará con ellos para tratar de descubrir qué algoritmos de recomendación funcionan mejor sobre qué sistemas de computación distribuida, tratando de encontrar los puntos débiles y fuertes de cada uno de ellos.

**Palabras clave:** Machine Learning, MapReduce, Pregel, Hadoop, Computación Distribuida, HDFS, Sistemas de Recomendación

# Abstract

Distributed file systems and distributed computation systems are gaining more importance nowadays as the data volumes continue increasing each day.

In the same way, recommendation systems are present in many different ways trying to improve the user experience by increasing their satisfaction.

This work tries to join this two fields of the computer engineering, by implementing four different recommendation algorithms on two distributed computation systems: MapReduce and Pregel.

After explain how this implementations work on the different systems and their particularities we will perform experiments on them trying to conclude if they are suitable to this kind of systems or not.

**Keywords:** Machine Learning, MapReduce, Pregel, Hadoop, Distributed Computation, HDFS, Recommendations Systems

# Índice

# Tabla de ilustraciones

# 1. Introducción

## 1.1 Motivación

El presente trabajo fin de grado tiene como motivación intentar determinar si los sistemas de recomendación, que están teniendo una importancia cada vez mayor en el día a día, y los sistemas de computación distribuida pueden convivir en un mismo ecosistema de tal modo que se obtenga una mejora sustancial en cuanto a velocidad, escalabilidad y eficiencia de los primeros.

A raíz del artículo de Google sobre el paradigma MapReduce (Dean, 2004) y cómo implementaron basándose en éste el Google File System (Ghemawat, 2003), Apache implementó su versión open source llamada Hadoop, convirtiéndose rápidamente en el sistema más utilizado para manejar grandes cantidades de datos.

Tanto Hadoop como GFS tienen ciertos problemas a la hora de manejar algunos datasets dependiendo de su taxonomía, por lo que de nuevo Google publicó en 2008 un artículo con lo que llamaron Pregel (Malewiczet et al 2010), un sistema para procesado de grafos de una manera distribuida.

Con motivo de entender y aprender dichos sistemas se propone este trabajo, de tal manera que el alumno pueda sumergirse en los diferentes sistemas de computación distribuida, entender los porqués de su nacimiento, así como sus ventajas e inconvenientes. De igual manera se quiere comprender los algoritmos de recomendación, sus formas de implementación, y las ventajas e inconvenientes que presenta su implementación sobre los diferentes modelos de computación distribuida.

## 1.2 Objetivos

Se tendrán como objetivos dos diferentes. En primer lugar la implementación de los principales algoritmos de recomendación sobre algunos de los sistemas de computación distribuida existentes actualmente, o en su defecto discernir que no son adaptables a tales modelos.

En segundo lugar, estudiar el efecto, sea positivo o negativo, de dicha implementación sobre estos sistemas en términos de efectividad, escalabilidad, y eficiencia de los algoritmos sobre los diferentes sistemas.

## 1.3 Estructura del Documento

Este documento se estructura de la manera que se describe a continuación. En el capítulo 2 se tratará el estado del arte de los sistemas de recomendación y de los sistemas de computación distribuida. En los capítulos 3 y 4 se explicarán las implementaciones de los algoritmos de recomendación elegidos basadas los sistemas de computación distribuida. A continuación se detallarán los experimentos llevados a cabo en el capítulo 6, y finalmente en el capítulo 7 se discutirán las conclusiones.

# 2. Estado del Arte

Tanto los Sistemas de Recomendación como los sistemas de computación distribuida o sistemas de archivos distribuidos son campos relativamente recientes pero presenten una gran variedad de implementaciones y diferentes formas de afrontar la problemática que cada uno pretenden resolver.

En los siguientes dos puntos se explicará cuál es el estado del arte para cada uno de ellos para lograr crear una visión general ambos, esto es, qué son, qué tipos hay, cuál es su estado de madurez actual, de donde vienen, y hacia dónde van.

## 2.1 Sistemas de Recomendación

Los Sistemas de Recomendación son herramientas que facilitan el acceso y descubrimiento a vastos catálogos de objetos (música, vídeo, libros, productos, etc.) de forma personalizada. El estudio y desarrollo de los Sistemas de Recomendación se encuentra en la confluencia de áreas como Aprendizaje Automático, Recuperación de Información o Modelado de Usuario. Un sistema de recomendación tiene como objetivo presentar al usuario objetos acordes a su gusto con la intención de recudir al máximo la tarea de exploración y descubrimiento por parte del usuario.

Los Sistemas de Recomendación son relativamente recientes pero han ganado una importancia sustancial en multitud de ámbitos. Los más comunes son los ámbitos relativos al consumo o compra de objetos como pueden ser películas, libros, música, o diferentes objetos de consuma de una tienda online, aunque cada vez se van extendiendo a más y más diferentes ámbitos como la recomendación de seguidores en redes sociales, restaurantes, servicios financieros u otros usuarios en sitios webs de citas.

### 2.1.1 Tipos de sistemas de recomendación

Debido a la gran variedad de ámbitos aplicables a la tarea de recomendación existe un gran espectro de sistemas, cada uno tratando de optimizar dicha tarea en base a las particularidades que presenta. Como es obvio, la recomendación de películas y la de restaurantes puede afrontarse de muy diferentes modos.

El tipo de información disponible sobre usuarios, objetos y la relación entre ambos determina el tipo de sistemas de recomendación aplicables a un dominio. En general, se distinguen las siguientes familias:

- Basados en contenido: tratan de analizar el contenido de los ítems para realizar las recomendaciones, tratando de acercar ítems con un contenido más relevante en base a los gustos del usuario.
- Filtrado colaborativo: utilizan los datos de interacción de usuarios con los ítems para encontrar patrones y filtrar los ítems interesantes para cada usuario.
- Basados en red social: utilizan la información recabada de una red social real tales como los ítems más populares, los ítems que gustaron a los amigos o los ratings que les dieron para extraer las recomendaciones.
- Híbridos: sistemas en los que se pretende aunar lo mejor de los sistemas anteriores combinando sus esfuerzos.

Los Sistemas de Recomendación basados en filtrado colaborativo son uno de las familias más estudiadas y empleadas debido a su generalidad (no dependen del contenido de los objetos) y su efectividad. En este trabajo nos centraremos en estas alternativas. Como se ha comentado antes, en el filtrado colaborativo las interacciones entre usuarios y objetos son usadas para generar recomendaciones a otros usuarios. Estas interacciones pueden ser **explícitas**, generalmente un rating proporcionado por el usuario a un ítem en concreto, o **implícita**s, como por ejemplo escuchar una canción, ver una película, etc.

Ambas formas de extraer la información tienen sus ventajas. La información explícita conlleva una interacción del usuario para expresar su opinión acerca de un ítem, cosa que, debido al esfuerzo cognitivo que ello implica, no siempre ocurre aunque se haya consumido dicho ítem. Por otro lado la opinión de este usuario puede estar sesgada o balanceada. Es decir, puede ocurrir que dos usuarios diferentes califiquen con un 80% a un ítem, pese a que a uno de ellos, muy exigente, le gustó mucho, mientras que al otro simplemente no le desagradó. Esta diferencia en la vara de medir de manera explícita puede generar que la tarea de recomendación no sea todo lo satisfactoria que se deseaba.

La recuperación de datos de manera implícita trata de recabar información del propio uso de los usuarios. Se puede tomar el simple hecho de que un usuario vio una película, o escuchó una canción. De tal modo que el hecho de que un usuario escuche una canción muchas veces, o todos los días, es un claro indicador de que le ha gustado, esto no ocurre con las películas o los restaurantes.

Así pues se puede entender que dependiendo del enfoque que se quiera dar al sistema de recomendación se puede uno decantar por un tipo de información u otra. En este trabajo **utilizaremos en nuestros algoritmos de recomendación el feedback implícito binario**, en el cual el valor de preferencia $r(u,i)$ entre un usuario $u$ y un objeto $i$ tomará los valores 1 y 0 dependiendo si el usuario ha interactuado con el ítem o no.

Teniendo en cuenta la gran cantidad de alternativas que existentes, en este trabajo hemos elegido los siguientes cuatro algoritmos: **user-based *k*-Nearest Neighbors, ítem-based *k*-Nearest Neighbors, Alternating Least Squares y probabilistic Latent Semantic Analysis.**

### 2.1.2 User-based *k*-Nearest Neighbors

Este algoritmo user-based *k*-Nearest Neighbors (ub-kNN)(Desrosiers et al, 2011)(Aiolli, 2013) basa sus recomendaciones en la similitud existente entre usuarios, presuponiendo que algo que le guste a un usuario $u$ es más probable que le guste a un usuario $v$ si ambos tienen gustos parecidos.

Para establecer cuanto de parecido son dos usuarios, este algoritmo define una medida de similitud entre usuarios que tiene en cuenta las preferencias de estos. Hay muchas y diferentes formas de calcular esta similitud. Una de las que mejor resultado ofrece y la que utilizaremos en las implementaciones de este trabajo es la similitud de Jaccard o coeficiente de Tanimoto:

$$sim(u,v) = \frac{|I_u \cap I_v|}{|I_u \cup I_v|}$$

donde $I_u$ indica los objetos con los que el usuario $u$ ha tenido alguna interacción.

Una vez calculadas las similitudes de los usuarios se procede a calcular las recomendaciones para cada usuario $u$. Para ello, se calculará un score para cada objeto $i$ que determina la predicción de preferencia. Dicho score se calculara agregando las

similitudes con el usuario objetivo $u$ con usuarios (los vecinos) que hayan tenido interacción con el objeto $i$:

$$s(u\,,i) = \sum_{v \in U_i} sim(u,v)$$

donde $U_i$ indica el conjunto de usuarios que han tenido interacción con el usuario.

Una variación del anterior algoritmo que proporciona mejoras en la eficiencia y efectividad del algoritmo consiste en considerar sólo las similitudes de un conjunto reducido de usuarios, el llamado vecindario, que generalmente consiste en el conjunto $N_k(u)$ de los $k$ usuarios más similares con el usuario objetivo $u$. De este modo, el cálculo del score quedaría así:

$$s(u,i) = \sum_{v \in U_i \cap N_k(u)} sim(u,v)$$

### 2.1.3 Item-based *k*-Nearest Neighbors

Este algoritmo, al contrario que ub-kNN, calcula similitudes entre objetos para generar recomendaciones. La idea es que, si dos objetos son parecidos, un usuario que tenga preferencia por uno de ellos probablemente también la tendrá por el otro (Desrosiers et al, 2011)(Aiolli, 2013).

El cálculo de similitudes entre objetos se realiza de forma similar al de usuarios en ub-kNN. En este caso la similitud de Jaccard entre ítems es la siguiente:

$$sim(i,j) = \frac{\left|U_i \cap U_j\right|}{\left|U_i \cup U_j\right|}$$

donde $U_i$ es el conjunto de usuarios que han interactuado con el objeto $i$.

El cálculo del score utilizará estas similitudes para recomendar los ítems más parecidos a los ya vistos por el usuario. De tal manera, para un ítem $i$ su score resultará de la suma de similitudes con los ítems ya conocidos por el usuario:

$$s(u\,,i) = \sum_{j \in I_u} sim(i,j)$$

Aunque menos usual, también se pueden obtener mejoras en eficiencia y efectividad al considerar los vecindarios $N_k(j)$ de los objetos del perfil del usuario:

$$s(u,i) = \sum_{j \in I_u, i \in N_k(j)} sim(i,j)$$

### 2.1.4 Semántica latente

La semántica latente aporta un cambio de visión respecto a los algoritmos anteriores, puesto que intenta basar la recomendación en los "factores" que hay detrás de cada usuario y objeto en el proceso de recomendación.

El modelo típico en semántica latente trata de modelar un espacio vectorial de dimensión $k$, el llamado espacio de factores latentes, que asocia a cada usuario $u$ con un

vector de factores $x_u \in \mathbb{R}^k$, y a cada objeto $i$ con un vector $y_i \in \mathbb{R}^k$. La predicción se calcularía multiplicando el vector de ítem por el de usuario:

$$s(u,i) = x_u \cdot y_i$$

Existe una gran variedad de métodos basados en semántica latente. En este trabajo, no centramos en dos variantes: **Alternating Least Squares** y **probabilistic Latent Semantic Analysis**.

#### 2.1.4.1 Alternating Least Squares (ALS)

Los métodos Alternating Least Squares consisten en minimizar una función de error cuadrático de las predicciones con respecto a los datos de entrenamiento alternando la minimización para los vectores de usuario e ítem. En esta sección procedemos a describir la alternativa de Hu et al. para datos implícitos (Hu et al, 2008).

Se tendrá a $r_{ui}$ como la variable binaria que indica si el usuario $u$ consumió el ítem $i$. También se presenta una variable $c_{ui}$ que representará la confianza en dichos valores, puesto que podría ser que alguien comprase un ítem para regalar a alguien sin gustarle el ítem en sí mismo, o que escuchase una canción porque la seleccionó sin querer y se le olvidó cambiarla. Esta variable crecerá a medida que $r_{ui}$crece y una posible medida es:

$$c_{ui} = 1 + \alpha r_{ui}$$

De este modo se tiene una mínima confianza en cada factor que irá aumentando según se tenga más información del usuario. Utilizando las notaciones de $R \in \mathbb{R}^{U,I}$ para la matriz de ratings de usuarios a ítems, $P \in \mathbb{R}^{U,K}$ para la matriz de factores de usuarios, $Q \in \mathbb{R}^{I,K}$ para la matriz de factores de ítems y $C \in \mathbb{R}^{U,I}$ para la matriz de valores de confianza.

Se tratará por tanto de encontrar el vector de factores que mejor defina al usuario minimizando por tanto la siguiente función de pérdida:

$$\min_{x*,y*} \sum_{u,i} C_{ui}(P_u Q_i^T - R_{ui})^2 + \lambda \left( \sum_u \|P_u\|^2 + \sum_i \|Q_i\|^2 \right)$$

El término dependiente de $\lambda$ es necesario para paliar el overfitting de los datos de entrenamiento. El algoritmo principal fijará una de las dos matrices para calcular la otra matriz que minimice dicha función, para después fijar ésta y calcular la otra durante un número de iteraciones N.

Pseudocódigo de ALS

```
initialize P and Q at random
for i in 0 to N
    P = arg min_Q(P, Q)
    Q = arg min_P(P, Q)
end for
```

Para minimizar la matriz de usuario:

$$P_u \underbrace{\left( \sum_i C_{ui} Q_i^t Q_i + \lambda I \right)}_{A} = \underbrace{\sum_i C_{ui} R_{ui} Q_i}_{B}$$

que sustituyendo C:

$$P_u \left( \underbrace{Q^t Q + \lambda I}_{A_1^P} + \sum_{i:R_{ui}>0} (C_{ui} - 1) \underbrace{Q_i^T Q_i}_{A_{2i}^P} \right) = \sum_{i:R_{ui}>0} R_{ui} C_{ui} Q_i$$

De igual manera se haría con la matriz de ítems habiendo fijado la matriz de usuarios. Un pseudocódigo del algoritmo de minimización para el caso de los usuarios es el siguiente:

Pseudocódigo de minimización

$precompute\ A_1^P\ and\ A_{2i}^P$

**for** $u \in U$ **do**
  $A = A_1^P$
  $B = 0$
  **for** $i$: $R_{ui} > 0$ **do**
    $A = A + (C_{ui} - 1) A_{2i}^P$
    $B = B + R_{ui} C_{ui} Q_i$
  **end for**
  $P_u = solve(P_u A = B)$
**end for**

#### 2.1.4.2 Probabilistic Latent Semantic Analysis (pLSA)

En este algoritmo descrito por Hofmann (Hofmann, 2004) tiene a $Z$ como un variable que puede tomar un número finito y predefinido $k$ de valores, el número de factores latentes, y a θ siendo las distribuciones de $P(z|u)$ y $P(z|i)$, las probabilidades de un factor $z$ para un usuario $u$ y un ítem $i$, y por tanto para realizar las recomendaciones, se combinará la probabilidad que ocurrir un ítem $i$ y un factor $z$ a la vez y la probabilidad de que dicho factor agrade al usuario en cuestión:

$$P(y|u;\theta) = \sum_z P(i,z) P(z|u)$$

Para acertar lo más posible en las recomendaciones será necesario por tanto ajustar θ, siendo $\hat{\theta}$ la óptima, utilizando el algoritmo Expectation-Maximization (Hilton, 1998). Este algoritmo consta de dos pasos que son aplicados alternativamente. El paso de Expectación(E), se calculan las probabilidades basando en las probabilidades $P(i,z)$ y $P(z|u)$, y el paso Maximization(M), donde se trata de optimizarlas:

- Expectation:

$$Q^*\left(z;u,i\ ;\ \hat{\theta}\right) = P(z\ |u,i;\hat{\theta}) = \frac{\hat{P}(i,z)\hat{P}(z|u)}{\sum_{z'} \hat{P}(i|z')\hat{P}(z'|u)}$$

- Maximization:

$$P(i,z) = \frac{\sum_{\langle u,i\prime\rangle : i' = i} Q^*(z;u,i,\hat{\theta})}{\sum_{\langle u,i\rangle} Q^*(z;u,i,\hat{\theta})}$$

$$P(z|u) = \frac{\sum_{\langle u\prime,i\rangle : u' = u} Q^*(z;u,i,\hat{\theta})}{\sum_{z\prime}\sum_{\langle u,i\rangle : u' = u} Q^*(z';u,i,\hat{\theta})}$$

## 2.2 Sistemas de archivos distribuidos y computación distribuida

Según los volúmenes de información que se manejaban iban aumentando en su tamaño y variedad, los estándares utilizados empezaron a mostrar problemas para abarcar tal cantidad de información de la manera que se venía haciendo hasta el momento. Con el auge de internet, redes sociales, tiendas online, etc. los lugares de donde recabar información han ido aumentando a niveles insospechados y los volúmenes de datos lo hicieron por tanto en concordancia.

Por ello Google creo su Google File System (GFS), un sistema de archivos distribuido, tolerante a fallos y que pudiese abarcar volúmenes de datos del orden de petabytes. Inspirándose en la ideas del GFS, Apache implementó lo que rápido se impondría como la referencia en el manejo de datos a gran escala: Hadoop[1], donde el acceso a los datos se realiza gracias al paradigma conocido como MapReduce.

A su vez, para afrontar otros tipos de problemas en los que el algoritmo MapReduce no es aplicable apareció Pregel, centrado en el procesamiento de grafos a gran escala.

### 2.2.1 Hadoop y HDFS

Hadoop y su HDFS (Hadoop Distributed File System) (Borthakur, 2007) es la implementación open source de un ecosistema donde conviven un sistema de archivos distribuido y diferentes servicios que ayudan a interactuar con él y con la información que alberga y es desarrollado por Apache. Está diseñado para ejecutarse sobre hardware fácilmente accesible (aplicable a los estándares de tecnología en la actualidad), pues pretende aumentar la escalabilidad, disponibilidad y resistencia a fallos de un sistema de archivos capaz de manejar grandes volúmenes de datos utilizando equipos de especificaciones "modestas" y de bajo coste.

En las siguientes secciones se tratará de explicar de manera superficial su infraestructura y funcionamiento.

#### 2.2.1.1 Sistema de archivos

Para poder almacenar datos mayores que la capacidad individual de cada nodo del cluster, HDFS particiona los archivos en bloques de tamaño configurable (siendo 64 o 128 MB los valores más comunes) y los distribuye a lo largo de los diferentes nodos con un factor de replicación también configurable. Por defecto éste valor es 3, es decir, cada bloque contará con al menos otras dos réplicas en el cluster.

Las réplicas se distribuyen a lo largo del cluster de manera inteligente. Hadoop dispondrá las dos copias de cada bloque de la siguiente manera: una en un nodo del mismo rack, y otra en un nodo de diferente rack. Esto es así para minimizar la distancia a recorrer

[1] www.hadoop.apache.org

en caso de tener que mover los datos, y para asegurar que al menos una réplica sigue accesible en caso de que un rack entero falle.

El acceso a los archivos completos es transparente y el usuario no sabrá en ningún momento en donde está cada bloque del archivo al que está accediendo sino que lo tratará como en un sistema de archivos tradicional y de hecho cuenta con muchos de los comandos a los que ya estamos acostumbrados en Linux (ls, rm, cat..).

#### 2.2.1.2 Tolerancia a fallos

Si el fallo de hardware es muchas veces la norma y no la excepción, lo es aún más cuando se dispone de equipos de bajo coste. Hadoop tiene en cuenta éste hecho para que su sistema sea tolerante a los posibles fallos de hardware que pueda ocurrir.

Puesto que es más barato y cómodo mover la computación que los datos, Hadoop tratará de mover la ejecución de la tarea fallida allí donde estén los datos sobre los que ha de operar. Sólo en caso de no ser posible traerá los datos a la unidad de cómputo en vez de al contrario.

#### 2.2.1.3 Infraestructura

La infraestructura de Hadoop es un entramado muy complejo y en constante desarrollo, que se compone de los siguientes componentes:

- Los DataNode y el NameNode son los responsables de mantener los bloques de archivos en correcto estado. Los DataNodes son simples almacenes de datos, mientras que el NameNode es el único que tiene conocimiento de en qué DataNode está cada bloque y dónde están sus réplicas. Por lo general, hay un NameNode y tantos DataNodes como se quieran, siendo la capacidad total del HDFS la suma de las capacidades de todos los DataNodes.
- JobTracker y TaskTracker: de manera parecida a los dos anteriores, el JobTracker es el encargado de partir cada trabajo en diferentes trabajos más pequeños llamados tareas o task. El JobTracker se encargará también de distribuir estas tareas por los nodos con servicio de TaskTracker para su computación.

Lo lógico es que cada DataNode tenga también rol de TaskTracker para minimizar el viaje de datos por red y escritura en disco, pero no es obligatorio.

### 2.2.1 MapReduce

El algoritmo MapReduce también fue desarrollado por Google conjunto con el GFS. MapReduce es un algoritmo pensado para trabajar sobre un sistema de archivos distribuido. En él se distinguen dos unidades principales llamadas Mapper y Reducer que trabajarán conjuntamente para llevar a cabo las operaciones deseadas.

Al tratarse de un archivo distribuido, las información necesaria para el cálculo correspondiente pudiera estar en más de una parte del fichero, y éstas estar distribuidas a lo largo de más de un nodo del cluster, por lo que será necesario organizarlas de tal manera que toda la información necesaria para los cálculos correspondientes estén en la misma unidad de procesamiento. De ésta tarea de encargan los mappers, para que posteriormente los reducers procesen la información, ya organizada convenientemente, como corresponda.

#### 2.2.1.1 Map

La fase Map es la encargada de organizar los datos de tal manera que se puedan realizar operaciones sobre ellos sin necesitar el conjunto total de los mismos.

Map recibe los datos en bulk y dejará la salida organizada en forma de lista según unos **<Key, Value>** establecidos por el usuario. Habrá tantos Mappers como particiones de archivo entrantes.

#### 2.2.1.2 Shuffle

La fase Shuffle tomará la salida del Map y redireccionará los datos en base a su key, mandando todos los valores asignados a la misma key a la misma unidad de procesamiento.

De esta manera cada nodo del cluster podrá realizar las operaciones que estime necesarias sobre cada key de los datos y sus valores.

#### 2.2.1.3 Reduce

La fase Reduce iterará sobre todos los valores de cada key y tras realizar las operaciones dejará la salida en forma de **<Key, Value>.**

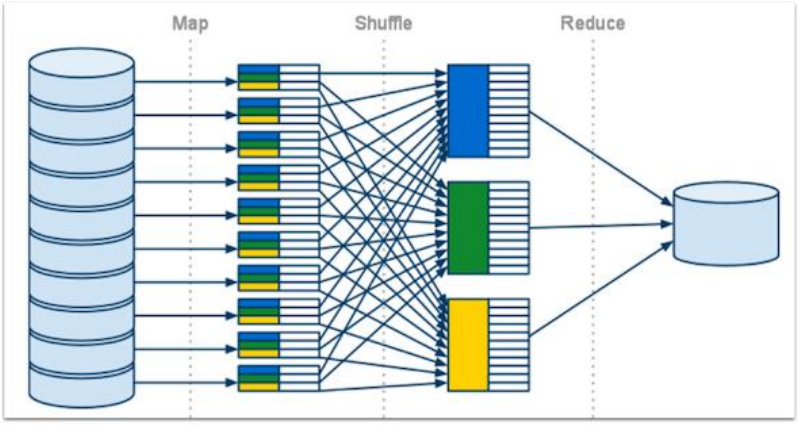



*Ilustración 1 Proceso MapReduce completo*

#### 2.2.1.4 Interfaces MapReduce

Las interfaces de Hadoop para MapReduce establecen que los mappers, tras procesar su partición de archivo correspondiente, dejarán una salida en forma de lista<Key, Value> que será la entrada de los reducers. Estos también dejarán como resultado otros <Key, Value> que pueden ser de mismo o diferente tipo. Los Key y Value deben implementar la interfaz Hadoop.io.Writable para que puedan ser escritos en disco entre paso y paso.

Es importante destacar que el Reducer recibe directamente un Iterable<Value> correspondiente a cada Key, por lo que la agrupación es transparente al programador.

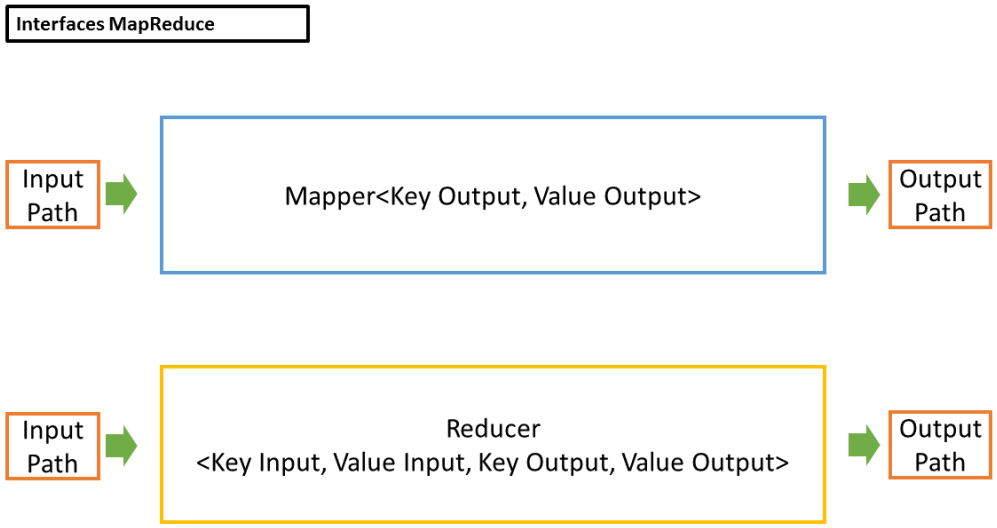



*Ilustración 2 Interfaces MapReduce*

### 2.2.2 Modelo Pregel

Pregel surgió como necesidad de afrontar cómputos a los que MapReduce no estaba capacitado convenientemente como es el cómputo de grafos de gran escala. Tras su publicación, Apache abrió el desarrollo de lo que llamaron Giraph, su implementación open source del modelo Pregel y que cabe destacar que se ejecuta sobre Hadoop y MapReduce.

De igual manera que el input de un algoritmo ejecutado sobre MapReduce es particionado y luego cada partición será enviada a cada una unidad de procesamiento, llamada Worker, para procesar cada partición por separado siendo todas coordinadas por un Master tal y como se muestra en la figura 4 y pudiendo comunicarse entre ellas por una serie de utensilios tales como mensajes, agregadores, o combinadores.

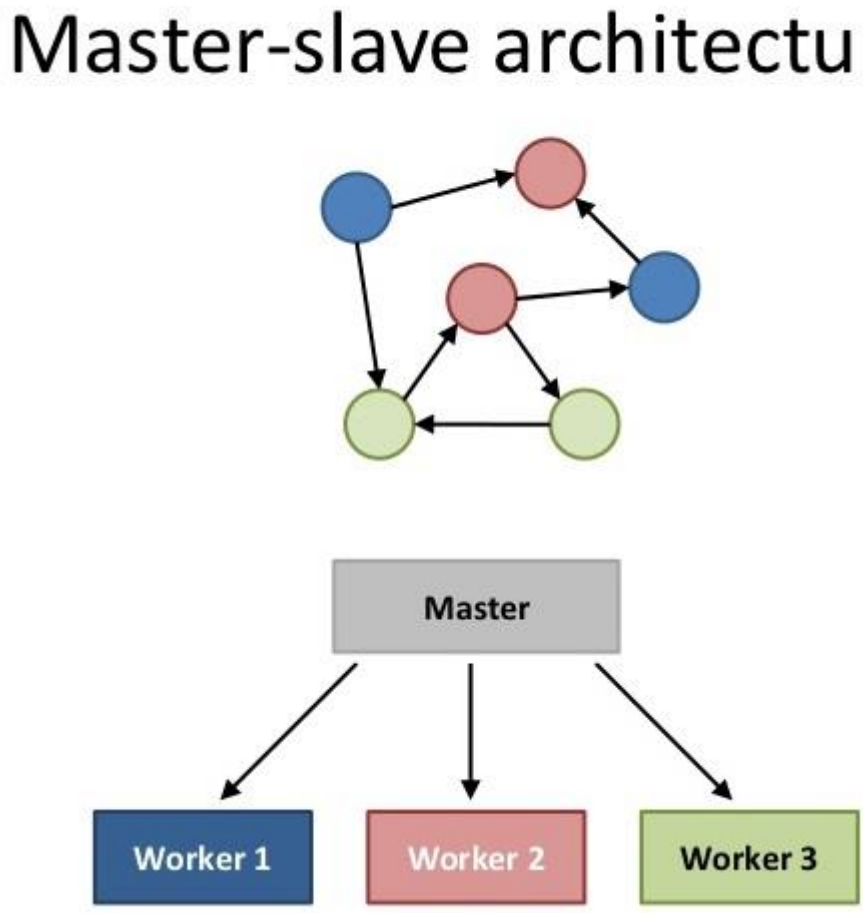
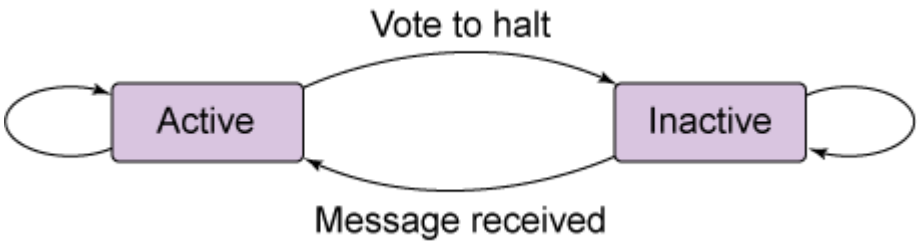



*Ilustración 3 Ejemplo de particionado y distribución de grafo en los diferentes workers*

La computación del grafo se enfoca desde el punto de vista de cada vértice. Cada uno de los vértices del grafo llevará a cabo su computación y se comunicará con los demás vértices del grafo por medio de mensajes. Cada ejecución de los vértices se denomina **superStep**. Tras cada superStep, cada vértice votará si ha de terminarse la computación. Cuando todos los vértices den por terminada su ejecución la computación global se detendrá.




*4 Sistema de votación en Pregel*

Las herramientas de las que dispone Pregel para la computación global y la comunicación entre vértices del grafo son las siguientes.

#### 2.2.2.1 Mensajes

Cada vértice podrá enviar mensajes a sus vértices adyacentes. El contenido de los mensajes dependerá del algoritmo y es cuestión del desarrollador.

Al inicio de cada superStep N, los vértices recibirán los mensajes que le fueron mandados por otros vértices en el paso N-1. Los mensajes de los pasos N y N-1 son independientes.

#### 2.2.2.2 Combinadores

Los combinadores tienen como labor reducir el número de mensajes mandados entre vértices. No están activos por defectos y han de ser escritos y habilitados por el programador en caso de ser necesarios.

Imaginemos que a un vértice le interesa solo la suma de todos los valores enteros de los mensajes que recibe. A sus efectos le es indiferente recibir un único mensaje con dicha suma, y es ésta la labor del combinador.

#### 2.2.2.3 Agregadores

Los agregadores son un mecanismo de comunicación global. Cada vértice puede proveer de un valor al agregador, que utilizará los valores de todos los vértices para reducirlos y ser pasados como valor agregado en el siguiente superStep.

Una posible aplicación podría ser el valor mínimo de alguna operación en todo el grafo. El agregador recibiría todos los valores del grafo y pasaría el valor mínimo a todos los vértices en el siguiente paso.

## 2.3 Librerías e implementaciones de sistemas de recomendación

En el ámbito de los sistemas de recomendación hay una gran variedad de implementaciones ya escritas para multitud de lenguajes de programación. Sin duda la más conocida es Mahout, una librería de machine learning desarrollada por Apache.

Mahout alberga multitud de algoritmos del campo del machine learning, algunos de ellos implementados para correr sobre el sistema de archivos distribuidos de Hadoop.


*Ilustración 5 Logo de Mahout*

Los algoritmos distribuidos ya implementados por Mahout y que utilizaremos tal cual están son el ItemBased y el ALS.

Para el procesado de grafos de forma distribuida contamos con Giraph, también desarrollado por Apache y que ofrece la implementación del modelo Pregel. Sobre Giraph corre la librería de Okapi, desarrollada por un equipo de telefónica y que implementa algoritmos de recomendación y de procesado y tratamiento de grafos. Okapi está todavía en una fase muy temprana y de los algoritmos mencionados anteriormente Okapi solo ha implementado el ALS.

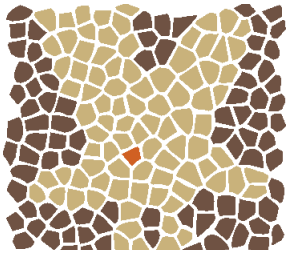


*Ilustración 6 Logo de Giraph*

*Ilustración 7 Logo de Okapi*

# *3.* Soluciones Basadas en el modelo *MapReduce*

En este capítulo proponemos implementaciones de los algoritmos de recomendación presentados en el capítulo 2 sobre el modelo MapReduce. En la librería Mahout[2] ya están disponibles implementaciones de los algoritmos ítem-based *k*-Nearest Neighbors y ALS, por lo que nuestro trabajo ha consistido en estudiar y describir el diseño de estos. Sobre los otros dos algoritmos abordados, user-based *k*-Nearest Neighbors y probabilistic Latent Semantic Analysis, tuvimos que diseñar implementaciones desde cero cuyo diseño explicaremos también en este capítulo.

Todos los algoritmos recibirán como input el path donde se encuentran los datos de los ratings dados por los usuarios a los ítems en formato [userID itemID rating].

Como ya hemos comentado, el rating se obviará puesto que trabajaremos con feedback implícito binario y a los efectos nos dará igual que haya puntuado alto o bajo.

## 3.1 Item-based *k*-Nearest Neighbors

El ítem based es implementado por Mahout y por tanto la labor llevada a cabo con él ha sido de comprensión y documentación. El cálculo de este algoritmo se divide en tres trabajos fases principales llamadas Prepare PreferenceMatrix Job, RowSimiliraity Job y Partial Multiply, que se explican por separado a continuación.

### 3.1.1 Prepare Preference Matrix Job

Dado que la entrada de los programas es un fichero con los rating en formato [userID, ItemdID, Rating], se necesita transformar esta entrada a una matriz donde se almacenen los vectores de usuarios y de ítems. Esta fase se encarga de preparar la matriz de ratings de las dos diferentes formas: una siendo usuarios las filas e ítems columnas, y su traspuesta, quedando así una matriz con los vectores de usuario y otra con los vectores de ítems.

ToItemPrefsVectorMapper-Reducer crearán los vectores de usuario, dejando la salida en forma de **<userID, userVector>.**

El mapper procesará el input [userID, itemID, rating] y dejará en la salida <userId, EntityPref>. EntityPref no es más que una encapsulación de [ItemID, Rating], puesto que la interfaz de mappers y reducers de Hadoop requiere solo un tipo de value por key. El reducer creará los vectores de usuario poniendo el valor del rating proporcionado por el EntityPref en la posición i-ésima indicada con i=ItemId.

Posteriormente ToItemVectorsMapper-Reducer crearán a partir de ellos los vectores de cada ítem y dejarán una salida de forma **<itemId, itemVector>.** El mapper tendrá como salida <itemId, itemVector> con un itemVector vacío en todos sus elementos excepto en el de un usuario. El reducer posterior se encargará de juntar todos los vectores de ítem, cada uno conteniendo un único valor, en un mismo vector con todos ellos.

[2] www.mahout.apache.org

| $\boldsymbol{ToItemPrefsVectorMapper}(\boldsymbol{u}, \boldsymbol{i}, \boldsymbol{r})$: | $\boldsymbol{SimilarityReducer}(\boldsymbol{u}, \boldsymbol{L}[\boldsymbol{Entities}])$: |
|---|---|
| E = $EntityPref < i, r >$<br>$write(u, E)$ | $Vector\ v$<br>$for\ E \in L[Entities]$:<br>$v[e.item] = e.rating$<br>$write(i, sims)$ |

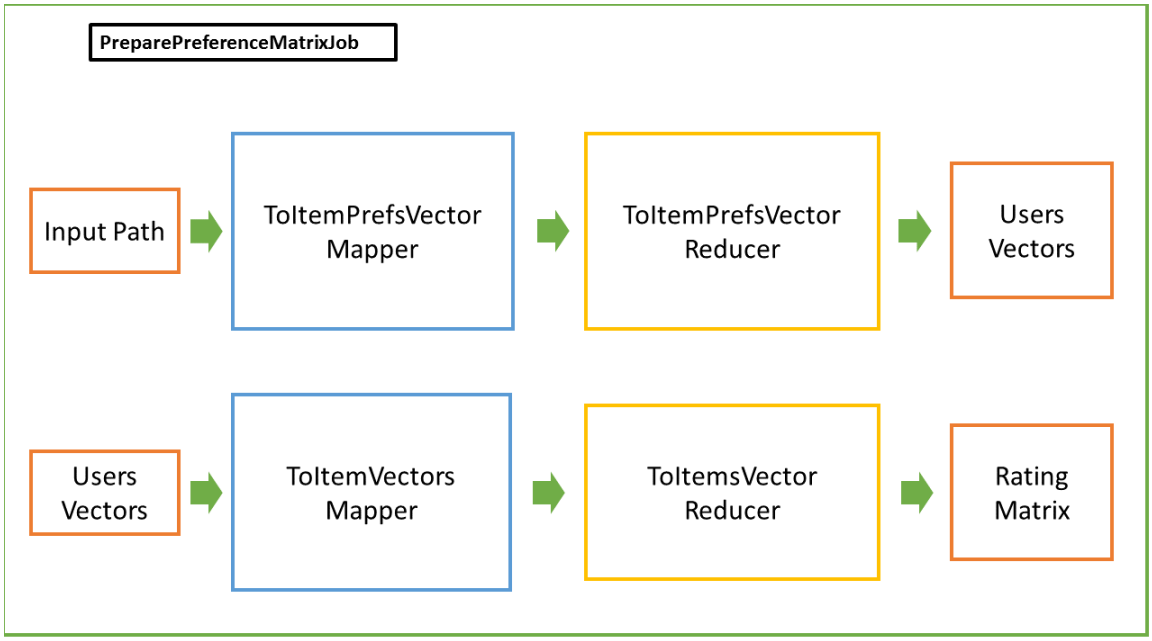



*Ilustración 8 Prepare Preference Matrix Job*

### 3.1.2 Row Similarity Job

Este trabajo será el encargado de calcular las similitudes entre ítems y seleccionar, para cada ítem, los otros *k* ítems más similares según se haya especificado en los parámetros de entrada.

Para ello recibirá los vectores de usuario y los recorrerá aumentando para cada ítem la intersección con los demás de su vector. Es decir, rellenará un vector de concurrencias, llamado *dots*, con las coincidencias de los pares de ítems.

| $\boldsymbol{CoocurrencesMapper}(\boldsymbol{u}, I_u)$: | $\boldsymbol{SimilarityReducer}(\boldsymbol{i}, \boldsymbol{L}[\boldsymbol{dots}])$: |
|---|---|
| $N = length(I_u)$<br>$for\ n = 0 \rightarrow N$:<br>$for\ m = n + 1 \rightarrow N$:<br>$dots[I_u[m]] = 1$<br>$write(I_u[n], dots)$ | $for\ dots \in L[dots]$:<br>$count = count + dots$<br>$for\ j \in count[j] > 0$:<br>$sims[j] = sim(i, j, count[j])$<br>$write(i, sims)$ |

Posteriormente el reducer agrupa todos los dots de cada ítem y calcula la similitud, dejando la salida en forma ItemID y vector de similitudes.

Tras el SimilarityReducer se obtienen la mitad de las similitudes, pues se habrá rellenado la mitad de una matriz que debería ser simétrica pues $sim(i,j) = sim(j,i)$. UnsimetrifyMapper se encargará de resolver este problema colocando en un nuevo vector las similitudes ausentes mientras que MergeToTopK reducer unirá todos esos vectores en uno solo y calculará cuales son las mayores K similitudes para posteriormente escribirlas en disco en forma de **<itemId, topKsimilitudes>.**

| $\boldsymbol{Unsymetriffy\ Mapper(i, sim)}$: | $\boldsymbol{MergeToTopKReducer(i, L[sim])}$: |
|---|---|
| $N = length(sim)$<br>$for\ n = 0 \rightarrow N$<br>$\quad write(n, sim[n])$<br>$write(i, sim)$ | $allSim = merge(L[sim])$<br>$topK = selectTopK(allSim)$<br>$write(i, topK)$ |

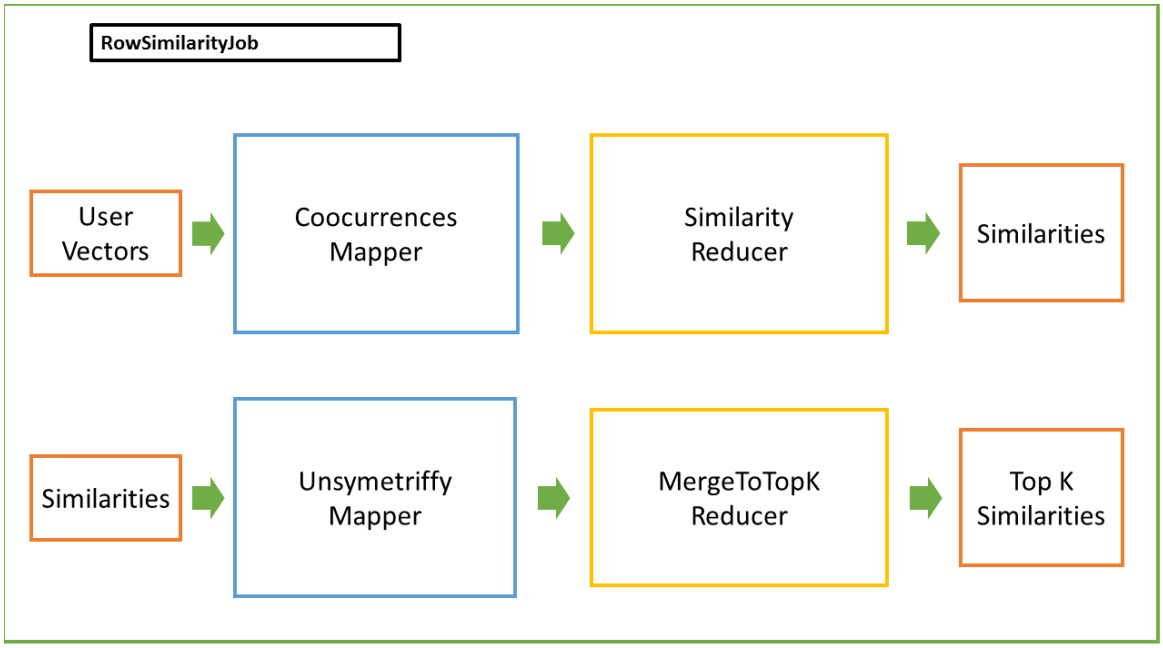



*Ilustración 9 RowSimilarityJob en ItemBased*

### 3.1.3 Partial Multiply

Esta fase calculará las recomendaciones para cada usuario. El primer paso de Partial Multiply recibirá como input dos paths conteniendo la matriz de similitudes y los vectores de usuario. El mapper creará una salida con <itemId, Preferencia> para cada uno de los valores del vector de usuario, y otra salida con <itemId, vector de similitudes> para cada ítem de la matriz de similitudes.

Como se debe utilizar solo un tipo de value para cada itemId tanto el rating como el vector de similitudes se encapsulará en otra clase, llamada VectorOrPref y que contendrá el vector de similitudes en un caso y el id de usuario junto con el rating que le dio al ítem en el otro.

Posteriormente el reducer los juntará creando una salida para cada itemId teniendo una clase VectorAndPrefs que contendrá la fila de la matriz de similitudes del ítem junto con todos los ratings dados por usuarios.

| | |
|---|---|
| $\boldsymbol{UserVectorSplitterMapper(u,v)}$: <br> $N = length(v)$ <br> $for\ n = 0 \rightarrow N$: <br> $Write(n, VectorOrPref(u, v[n]))$ | $\boldsymbol{ToVectorAndPrefReducer(i, L[V])}$: <br> $for\ VectorOrPref\ v \in L[V]$: <br> $if\ Pref$ <br> $VectorAndPref.setPref(v)$ <br> $else$ <br> $vectorAndPref.add(v.user, v.ratir$ <br> $write(i, VectorAndPref)$ |
| $\boldsymbol{SimilaritiesRowWrapperMapper(u,v)}$ <br> $write(i, VectorOrPref(v))$ | |

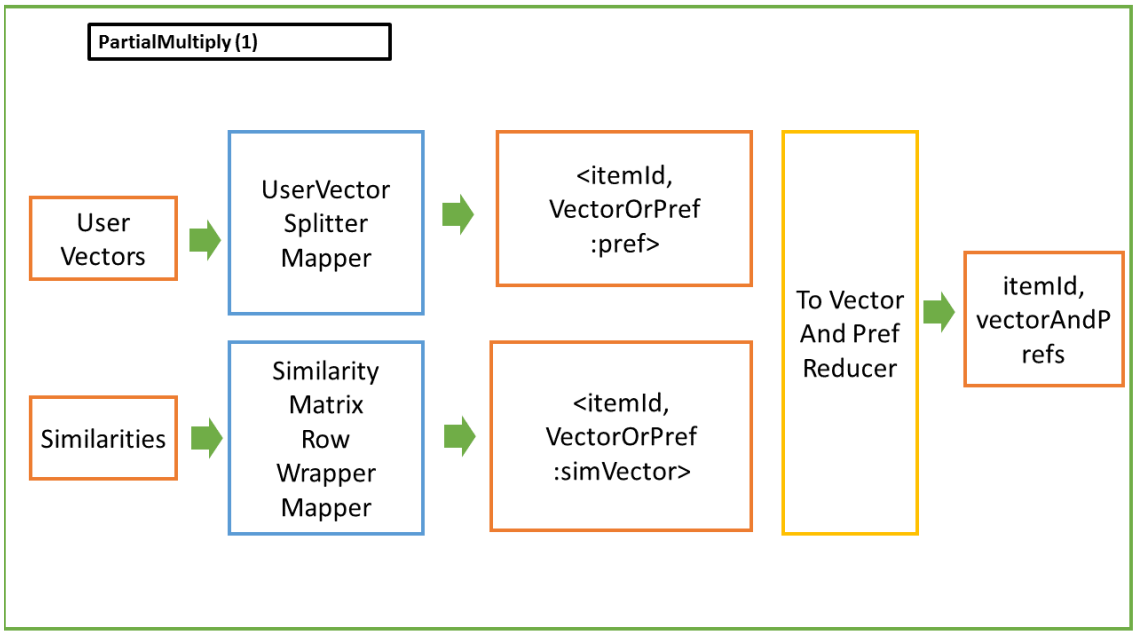



*Ilustración 10 Partial Multiply 1 en Item Based*

Con la salida anterior, se pasará a calcular la recomendación. Para ello el mapper recorrerá para cada ítem los vectores de usuario y dejará la salida en forma de <userId, PrefsAndSimilarities> done se encapsularán las preferencias a los ítems y el vector de similitudes. Finalmente el reducer procesará estos datos calculando la recomendación. En nuestro caso ya que utilizamos feedback implícito, bastará con sumar todos los vectores de similitud para obtener las recomendaciones.

| $\boldsymbol{PartialMultiply\ Mapper(i, vectorAndPref\ v)}$: | $\boldsymbol{AggregateAndRecomend\ Reducer(u, L[V])}$: |
|---|---|
| $N = v.numPrefs$<br>$for\ n = 0 \rightarrow N$:<br>$prefAndSimilarity(v.rating, v.simVector)$<br>$write(v.userAt(n), prefAndSimiliarity)$ | $for\ prefAndSimilarity\ \boldsymbol{p} \in L[V]$:<br>$recomendations+ = p.similarities$<br>$write(u, recomendations)$ |

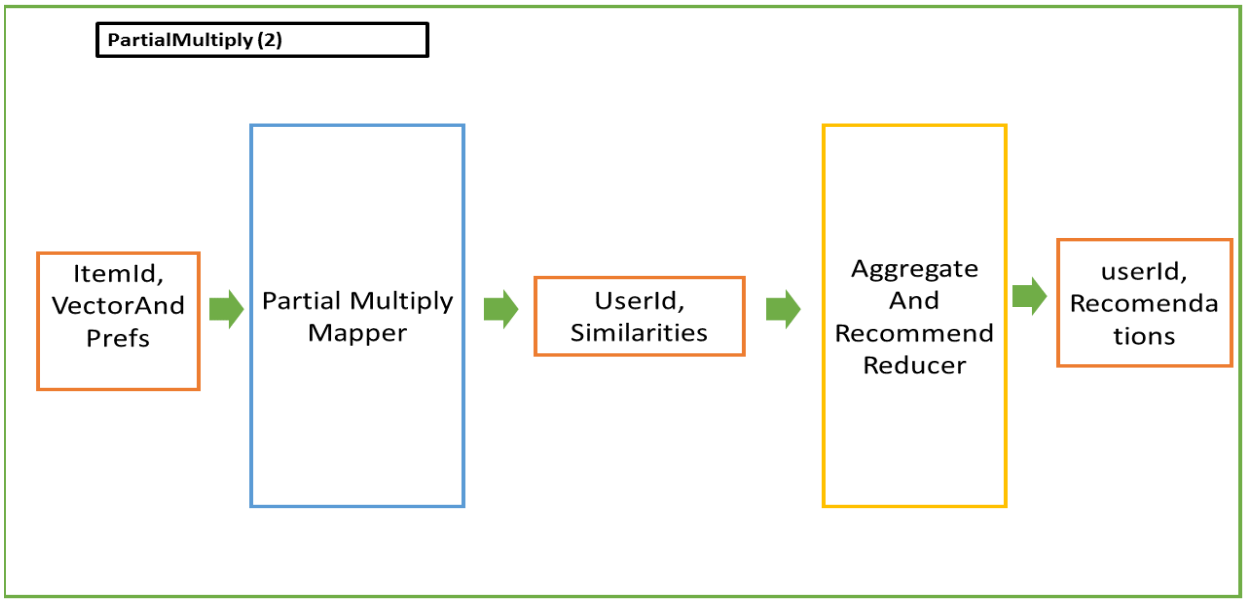



*Ilustración 11 Partial Multiply 2 en Item Based*

## 3.2 User-based *k*-Nearest Neighbors

Mahout no cuenta con un algoritmo basado en usuario por lo que se ha debido de implementar para este trabajo. Este algoritmo comparte muchos pasos con el ítem-based mostrado anteriormente. La primera fase en la que se preparan los vectores de usuario y de ítem es idéntica a la anterior por lo que se obviará en esta sección.

Las dos siguientes fases tienen ligeros cambios puesto que ahora se ha de calcular la similitud entre usuarios en lugar de ítems y la multiplicación final variará ligeramente.

### 3.2.1 Row Similarity Job

Mientras que en el anterior algoritmo este paso recibía los vectores de usuario para calcular la coocurrencia con los ítems que había puntuado, en éste caso será al revés y se recibirá los vectores de ítems para calcular la coocurrencia de los usuarios que le han visto.

| $\boldsymbol{CoocurrencesMapper(i, L[u])}$: | $\boldsymbol{SimilarityReducer(u, L[dots])}$: |
|---|---|
| $N = length(U_i)$<br>$for\ n = 0 \rightarrow N$: | $for\ dots \in L[dots]$:<br>$count = count + dots$ |

| | |
|---|---|
| $for\ m = n + 1 \rightarrow N:$<br>$dots[U_i[m]] = 1$<br>$write(U_i[n], dots)$ | $for\ j \in count[j] > 0:$<br>$sims[j] = sim(i, j, count[j])$<br>$write(u, sims)$ |

De igual modo que en el Item-Based el siguiente MapReduce cribará estas similitudes para seleccionar las top N similitudes.

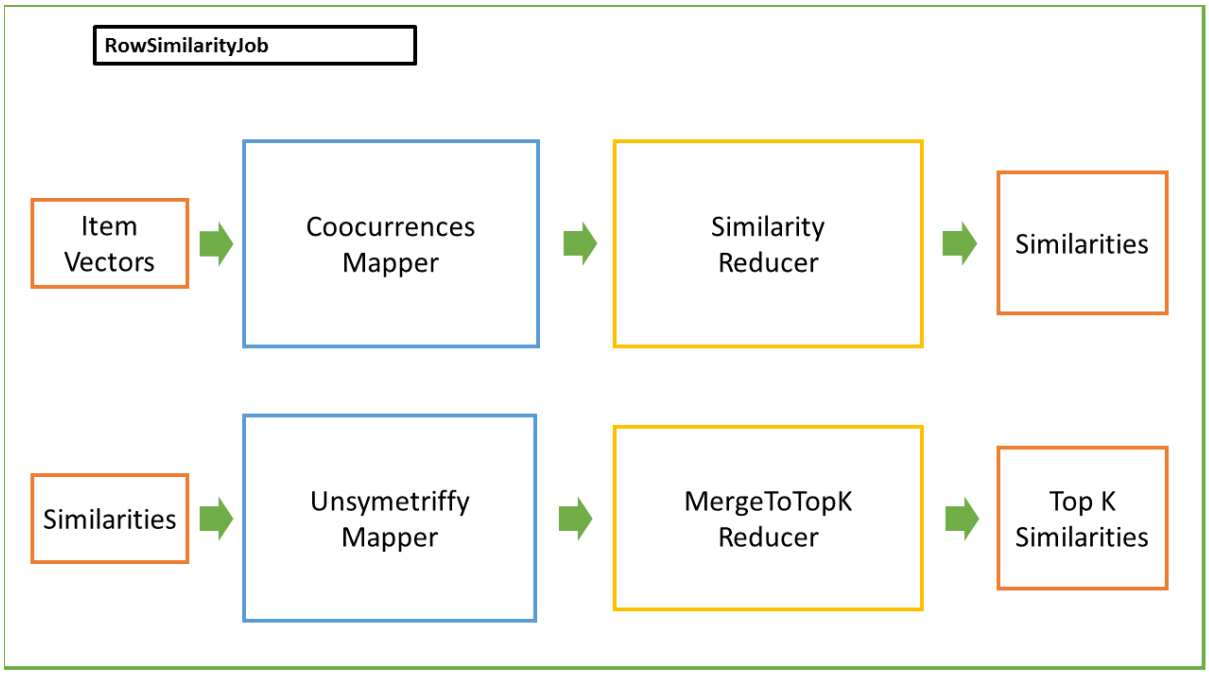



*Ilustración 12 Row Similairty Job en User Based*

### 3.2.2 Partial Multiply

Esta fase tendrá tres pasos en lugar de dos que tiene en el caso del ítem based. En primer lugar se recibirán las similitudes de usuario y los ratings de usuario, que se mapearán de manera parecida a como hicimos con los ítems. Cada fila de similitudes de usuario se mapeará directamente a una VectorOrPref mientras que los vectores de usuario se mapearán por cada ítem del vector creando un VectorOrPref para cada uno. De tal manera el map dejará una salida de modo <userId, VectorOrPref> que el reduce se encargará de juntar para dejar una salida del proceso MapReduce completo <userId, VectorAndPrefs>.

| | |
|---|---|
| $\boldsymbol{UserVectorSplitterMapper(u, v)}$:<br>$N = length(v)$<br>$for\ n = 0 \rightarrow N:$<br>$Write(u, VectorOrPref(n, v[n]))$ | $\boldsymbol{ToVectorAndPrefReducer(u, L[V])}$:<br>$for\ VectorOrPref\ v \in L[V]:$<br>$if\ Pref$<br>$VectorAndPref.setPref(v)$<br>$else$<br>$vectorAndPref.add(v.item, v.ratir$<br>$write(u, VectorAndPref)$ |
| $\boldsymbol{SimilaritiesRowWrapperMapper}$ **(u,v)**<br>$write(u, VectorOrPref(v))$ | |

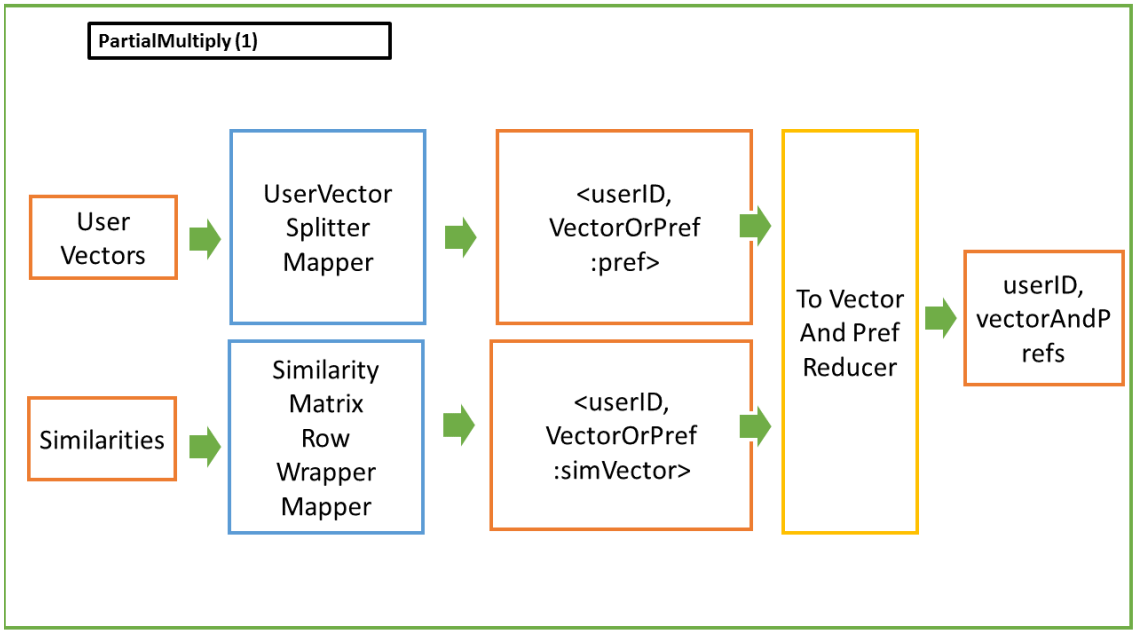



*Ilustración 13 Partial Multiply 1 en User Based*

En este momento tenemos las preferencias para cada usuario y sus similitudes con los demás usuarios. Claramente esto no nos sirve para recomendar puesto que de tal manera recomendaríamos los ítems que el usuario ya ha visto. Necesitamos por tanto trasponer esta matriz y por ello el siguiente paso se divide en dos.

De igual manera que en el ítem based se hará uso del PartialMultiplyMapper para escribir el itemId junto con su preferencia y el vector de similitudes del usuario, que el reducer se encargará de sumar.

Tras tener la suma de similitudes de usuarios, el siguiente MapReduce traspondrá la matriz quedando así las recomendaciones para los usuarios.

| | |
|---|---|
| $\boldsymbol{PartialMultiply}$<br>$\boldsymbol{Mapper}(\boldsymbol{u}, \boldsymbol{vectorAndPref\ v})$:<br>$N = v.numPrefs$<br>$for\ n = 0 \rightarrow N$:<br>$prefAndSimilarity(v.rating, v.simVector)$<br>$write(v.itemAt(n), prefAndSimiliarity)$ | $\boldsymbol{VectorAdditionReducer}(\boldsymbol{i}, \boldsymbol{L}[\boldsymbol{V}])$:<br>$for\ prefAndSimilarity\ p \in L[V]$:<br>$recomendations += p.similarities$<br>$write(i, recomendations)$ |
| $\boldsymbol{TrasposeMapper}(\boldsymbol{i}, \boldsymbol{Recommendations})$ :<br>$N = recommendations.length$<br>$for\ n = 0 \rightarrow N$:<br>$vector.set(i, recommendations[n])$<br>$write(n, vector)$ | $\boldsymbol{RecommendReducer}(\boldsymbol{i}, \boldsymbol{L}[\boldsymbol{V}])$:<br>$all = merge\ L[V]$:<br>$recomendations = select\ topK(all)$<br>$write(i, recomendations)$ |

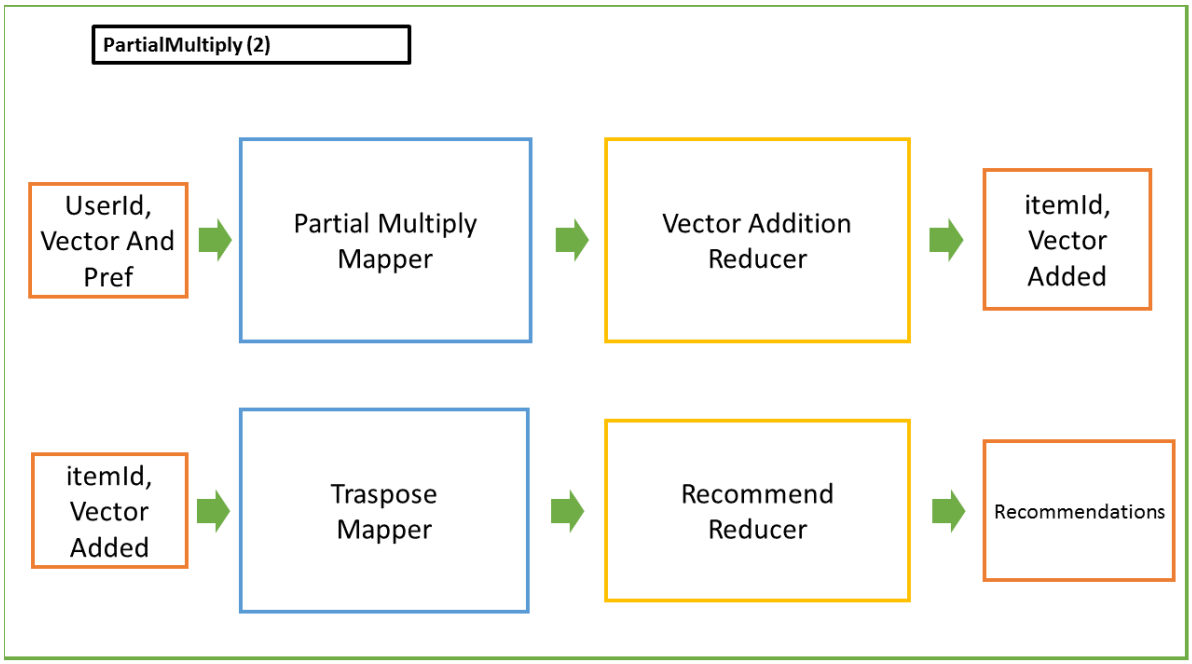



*Ilustración 14 Partial Multiply 2 en User Based*

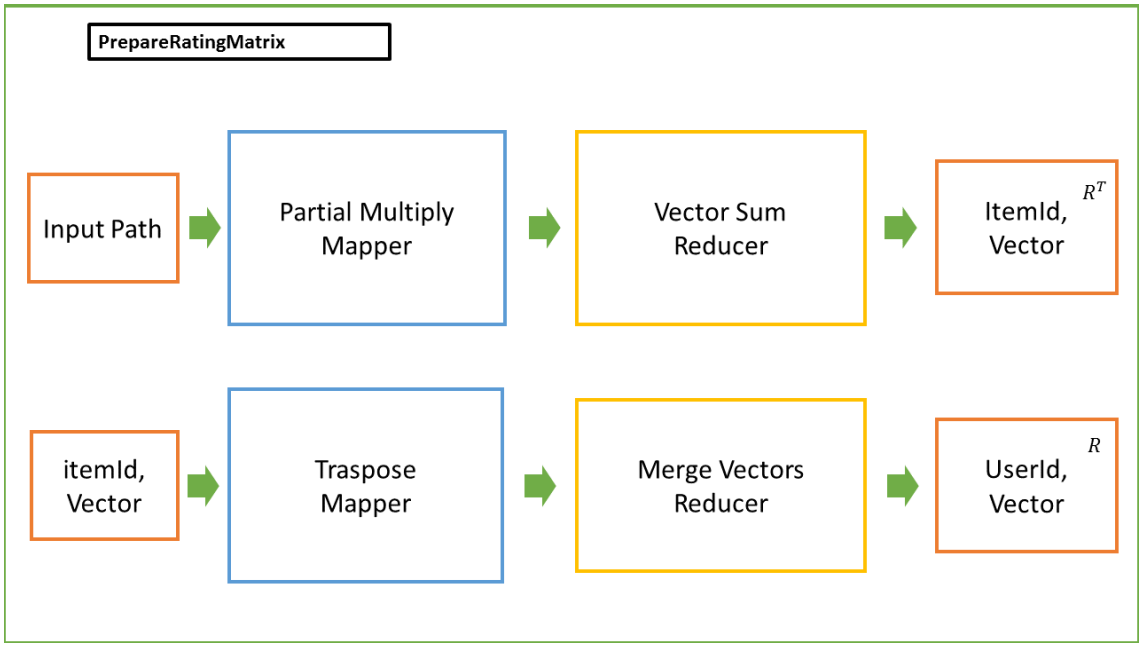
## 3.3 Alternating Least Squares

La implementación del ALS sobre MapReduce es la desarrollada por Mahout y se dividirá en una primera fase donde se calculará las matrices de factores y una segunda fase donde se llevará a cabo la recomendación.

### 3.3.1 Parallel ALS Factorization Job

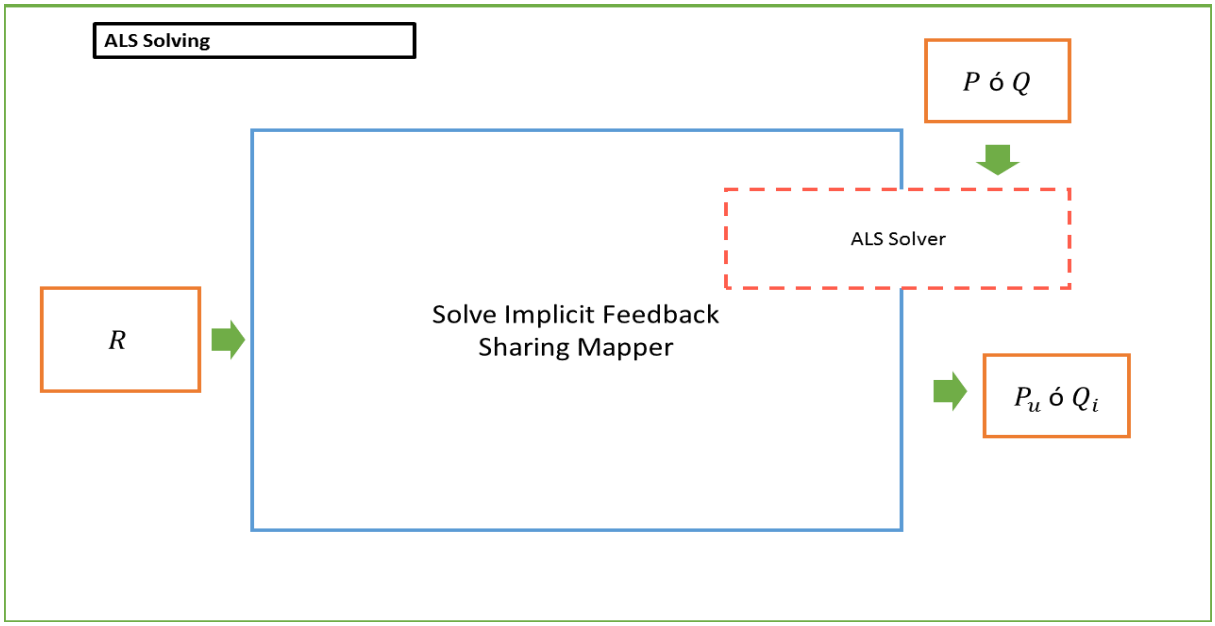
Esta primera fase es un proceso iterativo donde se realizaran los cálculos de $P$ y $Q$ según corresponda durante un número de iteraciones N definida a la hora de llamar al programa.

Como todos estos algoritmos, se recibirá como entrada el Path donde están contenidos los ratings de los usuarios, y en un primer paso se creará la matriz de vectores de usuario, $R$ y la matriz de vectores de ítems, que no es otra que $R^T$.



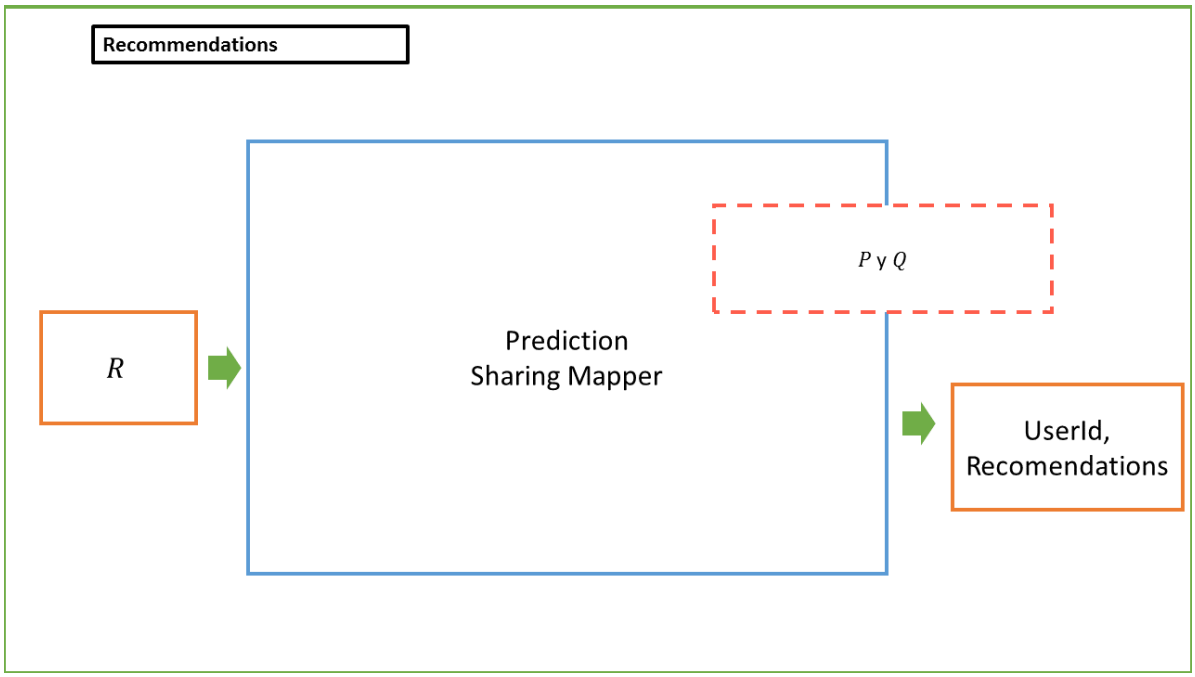

Seguidamente se inicializará la matriz de factores de ítems Q y se empezará a realizar las iteraciones del algoritmo. Recordemos que $P_u \underbrace{(\sum_i C_{ui} Q_i^t Q_i + \lambda I)}_{A} = \underbrace{\sum_i C_{ui} R_{ui} Q_i}_{B}$, por tanto en cada iteración necesitaremos $R$, y $P$, y $Q$. Los vectores $P_u$ y $Q_i$ se pueden calcular paralelamente, puesto que son idependientes. Por tanto cada nodo puede calcular unos vectores por separado y unirlos a la salida.

En la primera iteración donde se calcularán los vectores $P_u$se necesitará como entrada a $Q$ y $R$. Esto es un problema en el ecosistema de Hadoop puesto que está preparado para recibir uno o varios paths, pero tratarlos por separado. Para resolver este obstáculo se requerirá a una clase ALSSolver, encargada de resolver la computación de $P \cdot A = B$. Esta clase se instanciará como una clase compartida entre todos los mappers, que a su vez recibirá $Q$ o $P$ según corresponda por medio de una caché distribuida para todos los mappers. De ésta forma se podrá tener por una parte la matriz $R$ y por otra $Q$ o $P$. No se necesitará reducer puesto que el mapper calculará el vector completo.



### 3.3.2 Recommendation Job

Tras haber calculado las matrices de factores latentes de usuarios y de ítems, solo falta la propia tarea de recomendación en sí misa, donde se han de multiplicar los vectores de usuarios por los de ítems. La única dificultad radica de nuevo en que el paradigma MapReduce está pensado para procesar un directorio de entrada, y en este caso se requiere $P$, $Q$ y $R$. Para ello se volverá a hacer uso de un sharing mapper en el que se inclurián $P$ y $Q$ para complementar a $R$ que se introducirá por la entrada estándar y que el Mapper se encargará de multiplicar.



# *4.* Soluciones Basadas en el Modelo *Pregel*

El modelo Pregel fue pensado para el procesado de grandes grafos de manera distribuida. Apache Giraph es una implementación open source de este modelo, que utiliza el ecosistema de Hadoop para su funcionamiento, esto es, se aprovecha del HDFS y del servicio MapReduce entre otros.

Se ha tomado como referencia la librería de Okapi, que implementa algunos algoritmos de recomendación y cálculo de métricas de grafos. El único algoritmo de recomendación que tiene con este trabajo es el ALS, del cual solo se utilizará el esqueleto como guía para implementar un ALS para datos implícitos, pues el original del Okapi no cubría esta versión. Los algoritmos de K-Nearest-Neighbours User-based e Item-based, así como el pLSA se implementarán desde cero sobre Giraph.

## 4.1 Conceptos previos

Como ya se explicó en la sección 2.2.2 el modelo Pregel enfoca ésta computación desde el punto de vista de cada vértice del grafo, y la interfaz **Vertex** es la principal interfaz en la cual se sustenta todo el esquema. Esta interfaz cuenta con un método **Compute** que será el ejecutado por cada vértice cuando le llegue el momento, y será en él donde el vértice votará por detener la computación global o no. La computación global se detendrá cuando todos los vértices del grafo hayan votado tal cosa o el Master así lo decida.

Ésta interfaz requiere de las especificaciones de los tipos I, V, y E. I será el tipo del id único para cada vértice, V será el tipo del valor que alberga el vértice en su interior. Y E será el tipo del valor que tienen los arcos o aristas que a él conectan.

La computación se lleva a cabo por paso, denominados superSteps. En cada superstep cada vértice llevará a cabo su proceso Compute y decidirá si votar para detener la computación o no. Los vértices podrán intercambiar información con sus vértices adyacentes por medio de mensajes. Estos mensajes serán implementados por el programador y tendrán como único requisito implementar la interfaz Writable de Hadoop. Estos mensajes serán recibidos por los vértices gracias a un iterador de mensajes en el siguiente superstep.

Para crear algoritmos de recomendación sobre este paradigma, se adaptarán los datos para forman grafos de tal modo que los usuarios y los ítems sean vértices y un rating o interacción entre un usuario y un ítem unirá dichos vértices por medio de una arista.

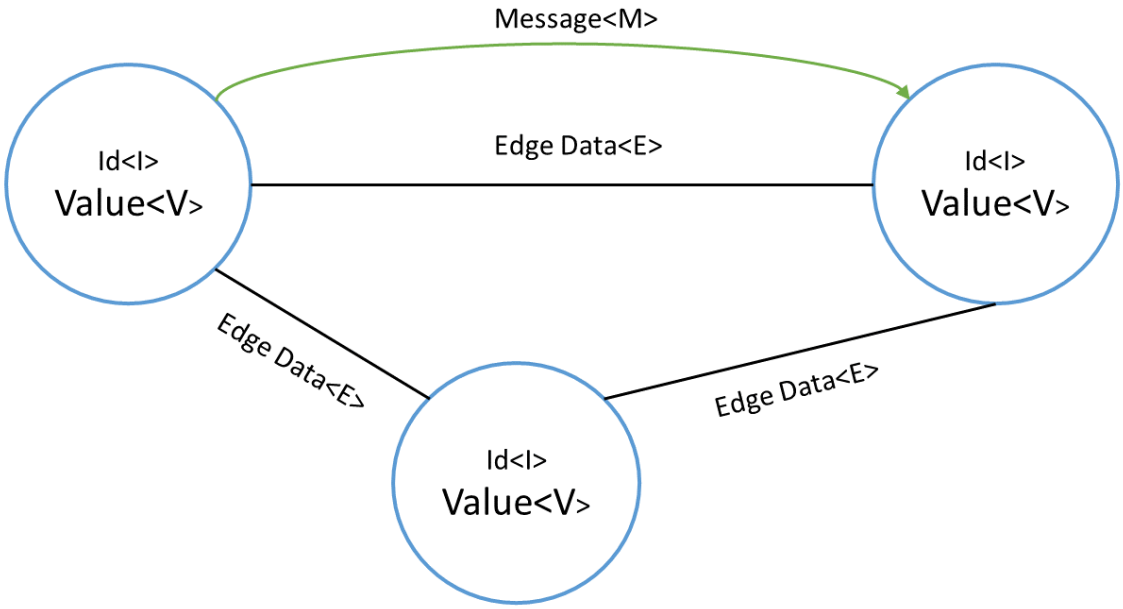



*Ilustración 15 Modelo Pregel*

## 4.2 Item-based *k*-Nearest Neighbors

Este algoritmo de recomendación necesitará 4 superSteps para realizar las recomendaciones, en el primero se enviará información de los ítems a los usuarios, estos reenviarán esta información a todos sus ítems para que calculen las similitudes, y posteriormente se enviarán éstas a los usuarios para que encuentren los mejores ítems para ellos.

Se utilizará como ID un id entero, y como Value una clase que albergará un vector de recomendaciones y los estados waiting y active. Los mensajes serán otra clase donde se podrá introducir diferente tipo de información como arrays de enteros, enteros o maps.

### 4.2.1 SuperStep 1

Como primer paso del algoritmo los usuarios que requieran recomendaciones mandarán a todos sus ítems una solicitud de recomendación. A su vez, los ítems mandarán a todos los usuarios un mensaje con información de su ID y su número de aristas, que no es otra cosa que el número de usuarios que le han consumido.

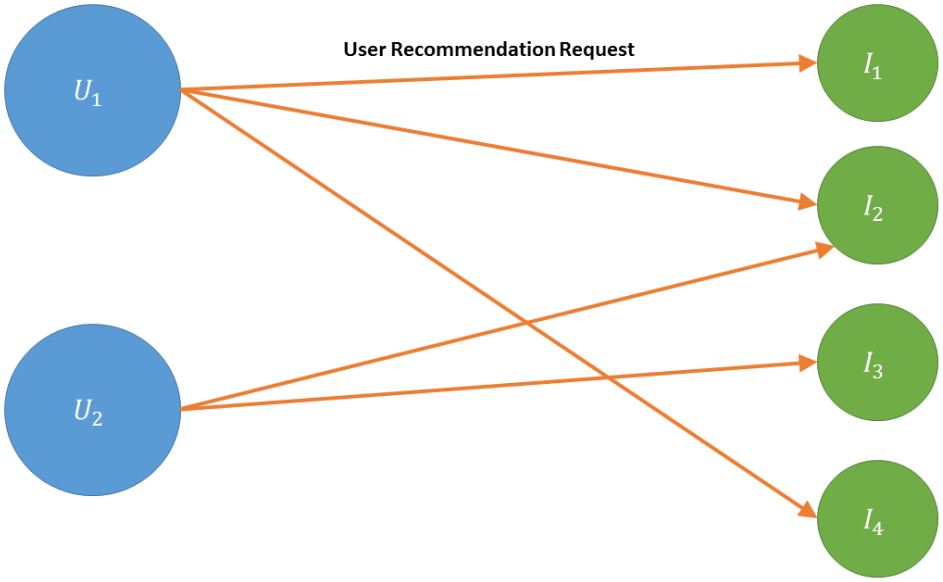



*Ilustración 16 Item Based superStep 1-1*

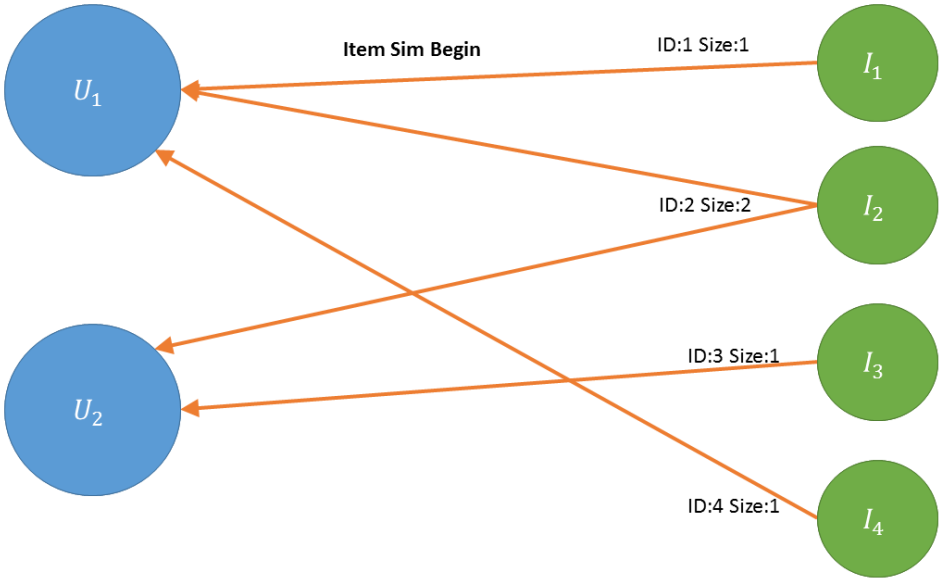



*Ilustración 17 Item Based superStep 1-2*

### 4.2.2 SuperStep 2

En este superstep cada usuario recibe mensajes de todos sus ítems con la información de su ID y tamaño. Cada usuario recopilará la información de los ítems y sus tamaños y se lo enviará a los ítems para que calculen la similitud que tienen con estos ítems.

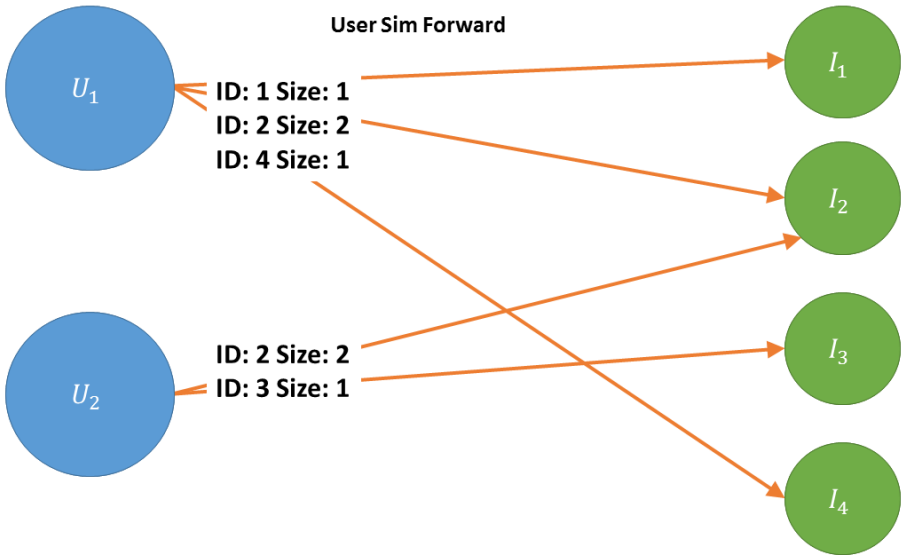



*Ilustración 18 Item Based superstep 2*

### 4.2.3 SuperStep 3

En éste paso será donde los ítems calculen las similitudes que tienen con los demás ítems. El número de usuarios que comparten un ítem $i$ con un ítem $j$ es el número de mensajes con información acerca de $j$ que le llegan. Tras observar esto, cada ítem tiene el tamaño de los ítems con los que tiene usuarios en común, cuántos tiene, y el tamaño del propio ítem (el número de aristas que posee el vértice), por tanto tiene todos los datos de unión e intersección para calcular la similitud: $sim(i,j) = \frac{|U_i \cap U_j|}{|U_i \cup U_j|}$

Tras calcular las N mejores similitudes las encapsulará en un mensaje que enviará a los usuarios que así lo indicaron en el superstep 1.

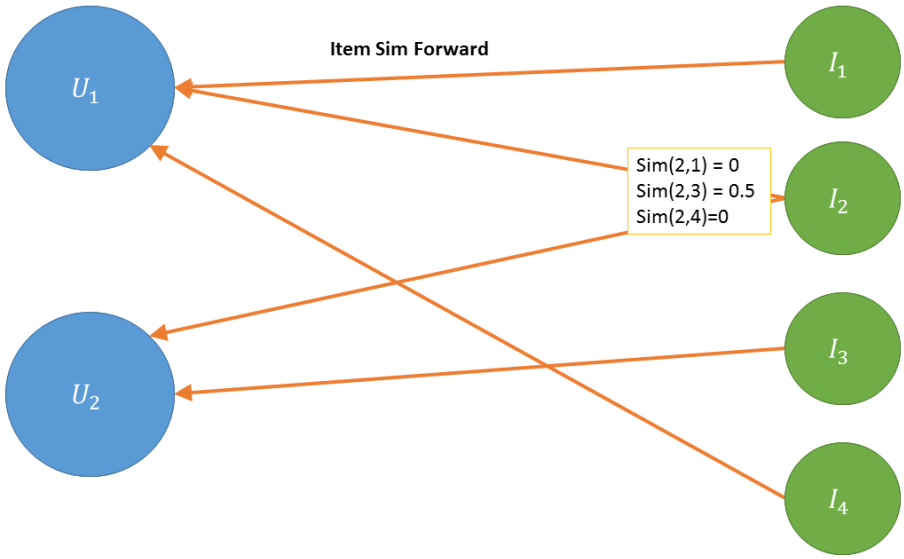



*Ilustración 19 Item Based superStep 3*

### 4.2.4 SuperStep 4

En el último paso, los vértices de usuario suman las similitudes de los ítems recibidos que no hayan consumido todavía: $s(u,i) = \sum_{j \in I_u, i \in N_k(j)} sim(i,j)$. Se obtiene la recomendación al elegir los N mejores ítems y votarán por detener la ejecución.


*Ilustración 20 Item Based superStep 4*

Los valores de los vértices de usuario serán los que se escriban en el fichero resultante de la ejecución, quedando éste con la información de cada ID de usuario y su recomendación.

## 4.3 User-based *k*-Nearest Neighbors

Este algoritmo requiere un superStep menos que el ítem based por lo que realizará su cálculo en 3 supersteps que se explican en las siguientes subsecciones.

### 4.3.1 SuperStep 1

Los usuarios mandan a los ítems mensajes conteniendo su ID y un array con los IDs de los ítems que han consumido.

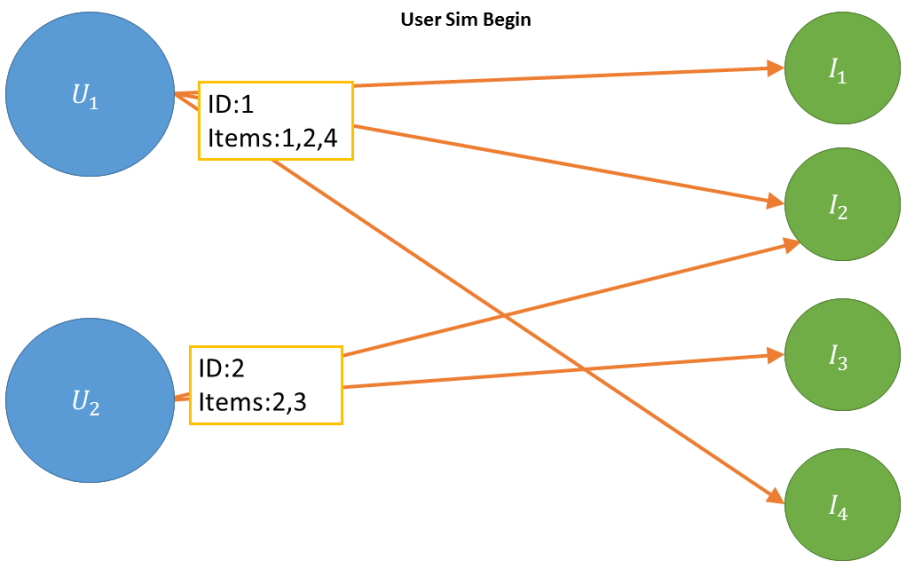



*Ilustración 21 User Based superStep 1*

### 4.3.2 SuperStep 2

Tras recibir la información de los ítems consumidos por cada usuario, los ítems volverán a enviar dicha información a los usuarios que le consumieron. En un primer momento se intentó hacer por medio de mensajes, pero la redundancia en la información y el número de mensajes hacía que no fuese posible realizar esta tarea sin aumentar el uso de RAM hasta sobrepasar los límites de memoria. Para solucionar esto, se utilizará un agregador. Con el uso de agregadores se eliminará la redundancia en la información: los ítems mandarán al agregador el ID de usuario junto con el array de ítems consumidos por éste. El agregador introducirá esta información en un Map<UserID, Array<ItemID>> de tal manera que se eliminarán los datos de usuario duplicados y la información estará disponible en el siguiente superstep.

A pesar de ello, los ítems mandarán mensajes a todos sus vértices adyacentes con información de cada usuario y el número de ítems que han consumido para que puedan calcular las similitudes con los user con los que tienen ítems en común en el siguiente superStep.

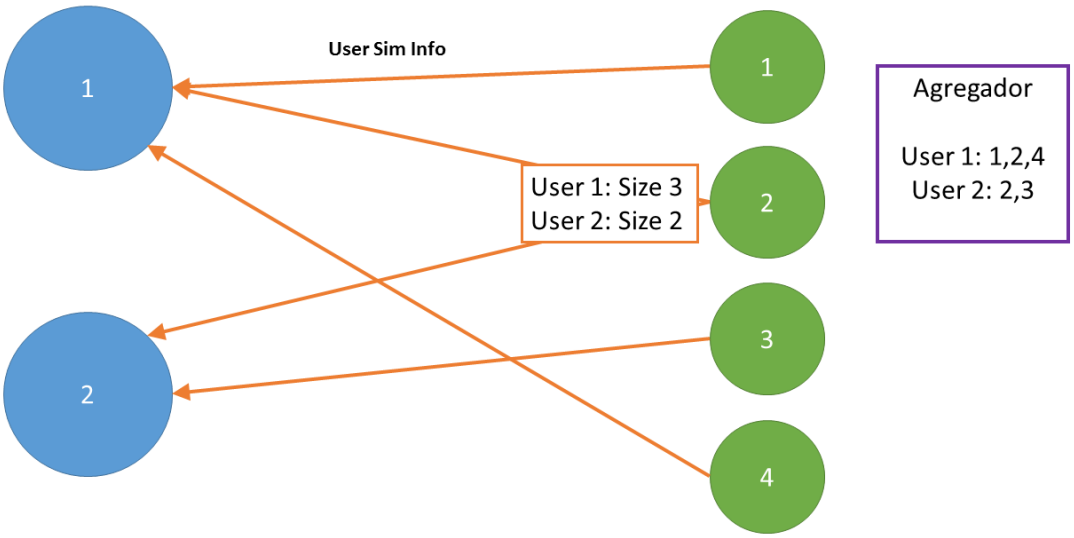



*Ilustración 22 Item Based superStep 2*

### 4.3.3 SuperStep 3

Este es el último superstep del algoritmo y es donde se conseguirán las recomendaciones. Cada usuario calculará su similitud con los datos de usuarios que ha recibido. Tras ello, observará en el agregador que ítems miraron los N usuarios más parecidos a él y sumará para cada ítemId la similitud con el usuario que le vio, obteniendo la recomendación tras elegir los top N ítems con mayor score. Se calculará en el método compute de cada vértice tanto $sim(u,v) = \frac{|I_u \cap I_v|}{|I_u \cup I_v|}$ como $s(u,i) = \sum_{v \in U_i} sim(u,v)$.

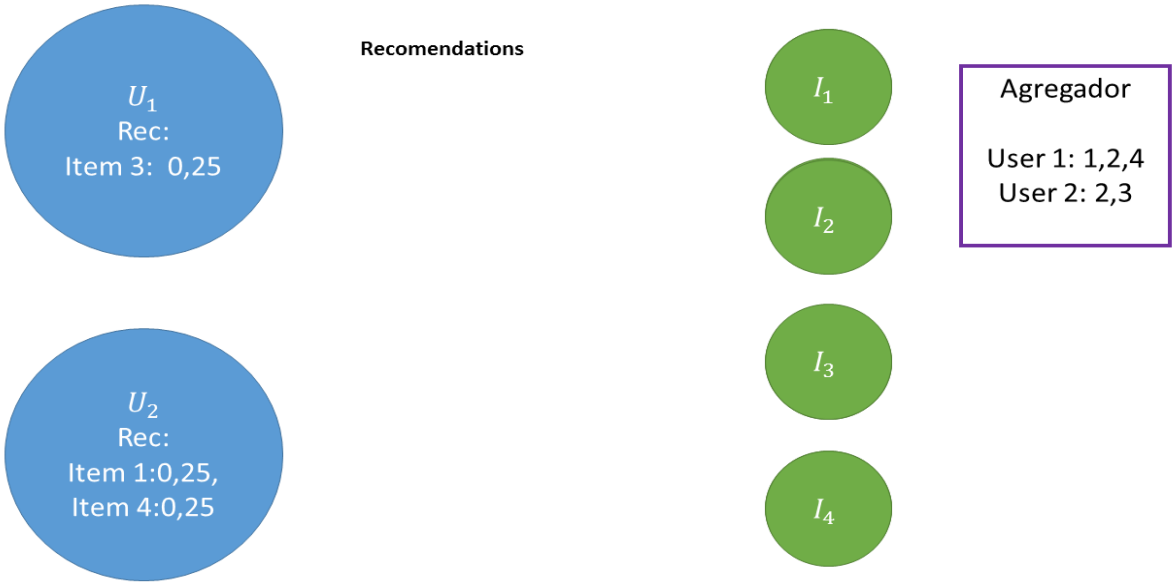



*Ilustración 23Item Based superStep 3*

## 4.4 Alternating Least Squares

El algoritmo de recomendación ALS con N iteraciones requerirá N+1 superSteps: N iteraciones del ALS más el superStep de inicialización de los vectores $P_u$ y $Q_i$. El grafo es el mismo que el implementado en los algoritmos KNN, pero cada vértice contendrá cada vector de usuario o de ítem de la matriz. En cada superStep se calculará por separado éste vector según corresponda.

Las aristas entre dos vértices del grafo indican que hubo interacción y representarán un valor de $r_{ui}$ por lo que serán necesarias para calcular $c_{ui} = 1 + \alpha r_{ui}$.

El punto importante desde el punto de vista computacional de este algoritmo se encuentra en el uso de un agregador donde se guardará la matriz fija del ALS ($P$ ó $Q$) y que cada vértice se encargará de enviar en cada superStep. De tal modo los vectores calculados en el superStep S estarán disponibles en el superstep S+1 por medio del agregador.

La infraestructura de Giraph nos permite definir una función llamada **preSuperStep** que como su nombre indica se ejecutará antes la ejecución de cada superStep y lo que es más importante: a nivel de worker. Recordemos que cada partición de un grafo se envía a un worker, de tal manera que cada worker asumirá la computación de los vértices a él destinados. Se puede aprovechar el método preSuperStep para calcular ciertos valores propios del ALS que se mantienen constantes para todos los vectores, pues dependen de la matriz fija, como es la matriz $A^P$ o $A^Q$.

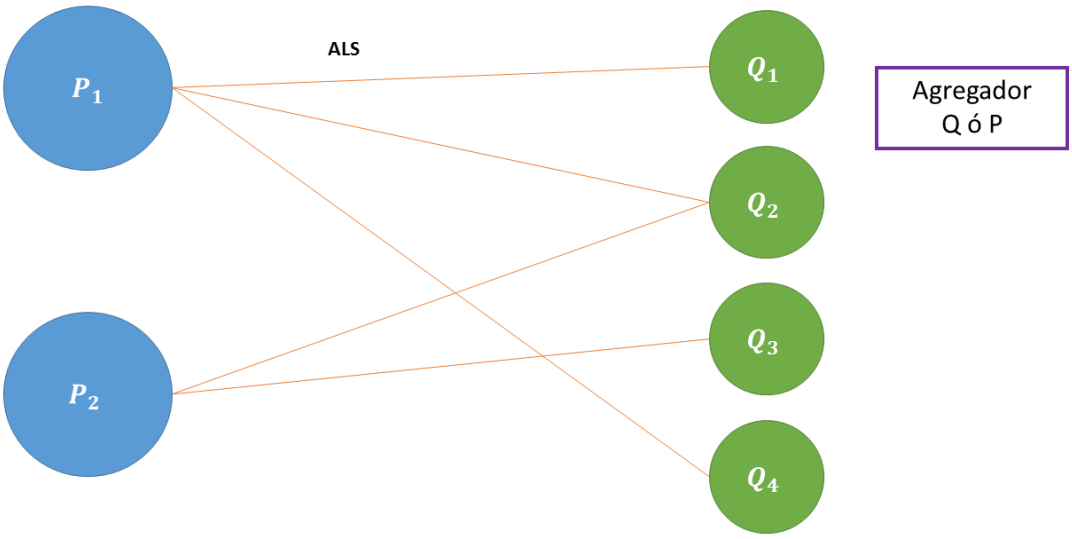



*Ilustración 24 Pregel -ALS*

La ejecución esta vez no terminará cuando todos los vértices hayan votado para tal cosa, sino cuando se hayan completado el número de iteraciones que se requerían para computar el ALS. Para controlar esto, se hará uso de la clase MasterCompute de Giraph, donde se puede indicar que se detenga la computación global en cualquier momento.

### 4.4.1 Recomendaciones

Al finalizar la ejecución se guarda en disco los vectores de $P$ y $Q$ que otro programa se debe de encargar de multiplicar según corresponda para conseguir las recomendaciones.

No es posible multiplicar en un modelo Pregel estos vectores, pues son pares usuario-item que no existen y por tanto no hay conexión entre sus vértices.

## 4.5 Probabilistic Latent Semantic Analysis

Este algoritmo pLSA es un algoritmo iterativo que consta de N iteraciones con 2 pasos por iteración, por lo que su implementación sobre el modelo de Pregel requerirá 2N+2 superSteps, estos dos últimos para la multiplicación de los vectores de usuario e ítem. De forma parecida a la implementación del ALS, los vértices de este grafo calcularan sus vectores $P(i,z)$ en el caso los items y $P(z|u)$ en el de los usuarios. Ambos vectores son los vectores necesarios para la calcular las recomendaciones pues recordemos que la recomendación, expresada en términos de probabilidad es $P(i|u;\theta) = \sum_z P(i,z)P(z|u)$.

Para optimizar dichos vectores utilizamos el algoritmo EM (Expectation-Maximization). En el primer superstep se inicializarán los vectores $P(i,z)$ y $P(z|u)$ con valores aleatorios, y los vértices de ítems mandarán un mensaje con su vector a los vértices de usuario, empezando así estos a calcular la fase de Expectation.

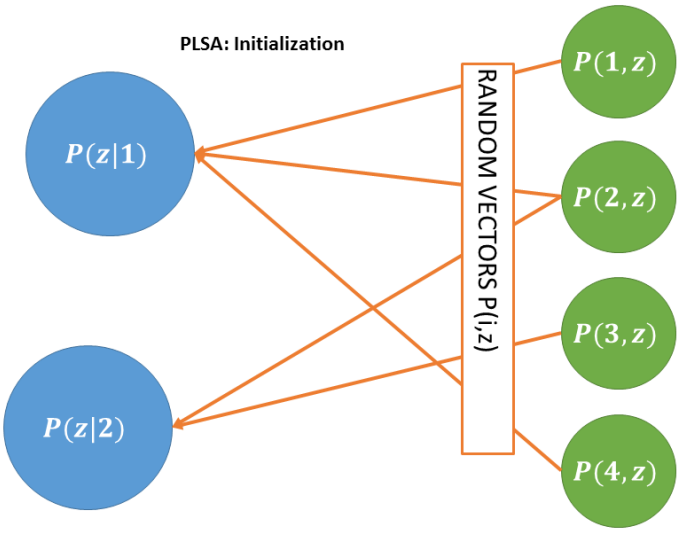



*Ilustración 25 Pregel - pLSA Initialization*

### 4.5.1 Expectation

Este paso se ejecutará en los vértices de usuario en todos los supersteps impares menores que 2N. Los vértices recopilaran los mensajes recogidos para calcular la fase de expectación y posteriormente:

(1) Colocan en el agregador $\sum P(z\,|u,i;\hat{\theta})$, que se encargará de sumar todos los que reciba y se tendrá a disposición en el siguiente superStep.
(2) Envían un mensaje conteniendo el $P(z|u\,,i)$ correspondiente a todos sus ítems.

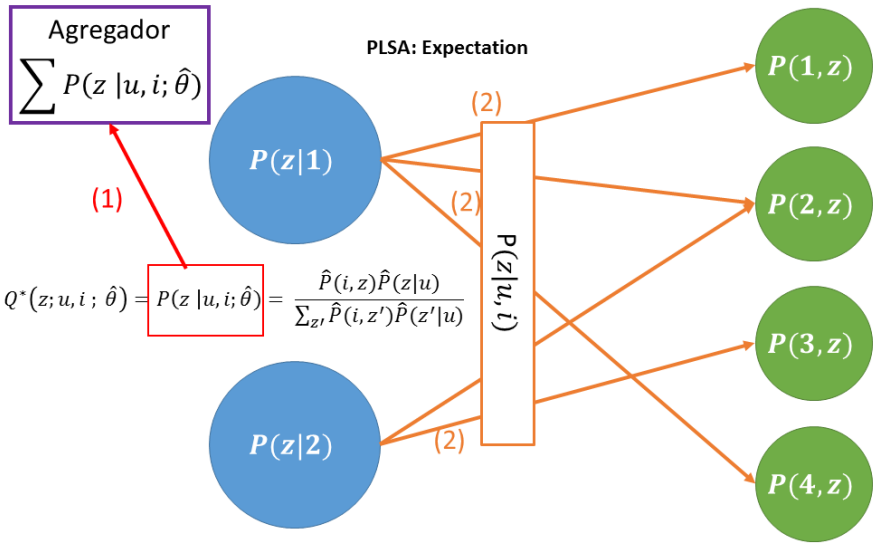



*Ilustración 26Pregel - pLSA Expectation*

### 4.5.2 Maximization

Esta fase se ejecutará en los dos tipos de vértices. En el vértice de los usuarios se llevará a cabo el cálculo de $P(z|u)$, mientras que en los ítems se calculará $P(i|z)$, que tras calcularlo se lo enviarán de nuevo a los vértices de usuario adyacentes para que calculen de nuevo el paso Expectation.

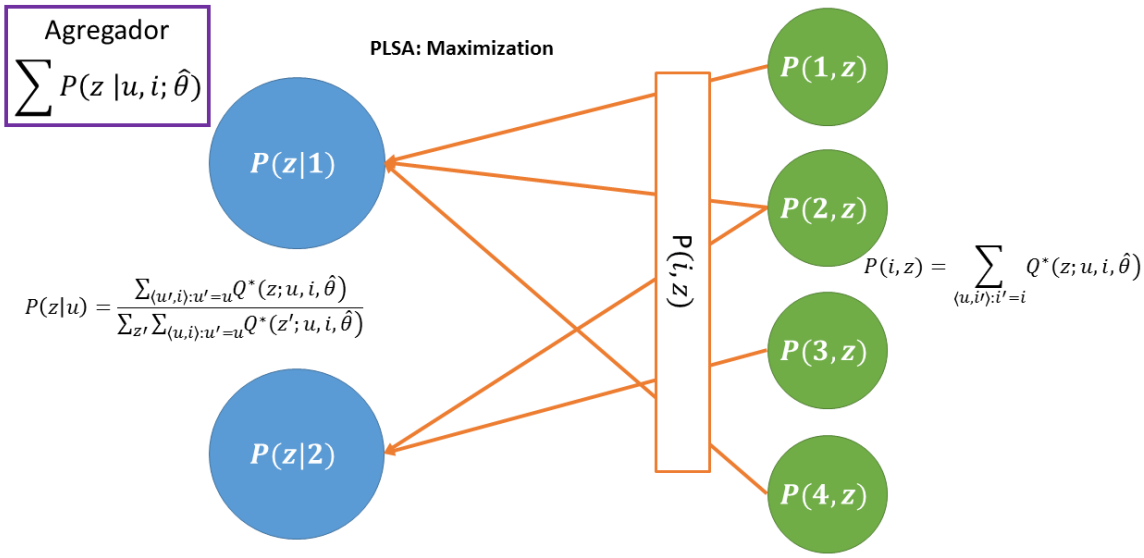



*Ilustración 27Pregel - pLSA Maximization*

### 4.5.3 Recomendaciones

Al finalizar se guardará en disco los valores de los vectores $P(i,z)$ y $P(z|u)$ que otro programa deberá encargarse de multiplicar según corresponda. Para calcular unas recomendaciones todos los usuarios deberían mandar un mensaje a los ítems que no han

visto, pero esto va contra natura del propio modelo de Pregel y, de hecho, no hay forma de hacerlo.

# 5. Experimentos

Tras implementar los diferentes algoritmos sobre los dos modelos es hora de experimentar para descubrir su comportamiento y sus ventajas e inconvenientes.

Para realizar las pruebas en este trabajo se monta un pequeño cluster de tres nodos con instalaciones de Cloudera[3]. Cada nodo tendrá dos núcleos y 8GB de memoria RAM. Uno de los nodos se utilizará exclusivamente como nodo master, puesto que la cantidad de recursos que consumen los servicios principales de Cloudera Manager hacen casi imposible el uso de ese nodo para otros aspectos.

Los algoritmos se probarán sobre el dataset de movieLens 1M, que consta de 1 millón ratings de 6000 usuarios a 4000 películas. Con nuestro análisis se pretende determinar que sistemas o algoritmos son mejores o peores para un entorno distribuido, qué algoritmos aprovechan más los recursos y disponibles y cuáles son las mejores opciones.

Se medirán valores de tiempo de ejecución, CPU utilizada y los valores máximos de escritura y lectura en disco y en red de cada algoritmo. Estos dos últimos son muy importantes en un cluster pues tanto la escritura simultánea como el viaje de datos por red pueden ser cuellos de botella en entornos reales.

## 5.1 Valores y parámetros de ejecución

Para la realización de los experimentos se han utilizado los siguientes valores de configuración de los mismos:

- En todos los casos se han calculado recomendaciones para todos los usuarios.
- Se han calculado 100 recomendaciones por usuario.
- En el caso de los algoritmos de KNN se ha establecido un vecindario de K=100.
- Para los algoritmos de la familia de semántica latente se ha establecido un número de factores latentes K=50.
- Para el ALS se ha determinado unos valores de $\lambda = 0.1$, $\alpha = 1$, y 20 iteraciones.
- Para el pLSA se han utilizado 20 iteraciones.

## 5.2 Experimentos sobre modelo MapReduce

En esta primera subsección se analizará el rendimiento de los algoritmos implementados sobre el paradigma de MapReduce.

### 5.2.1 Tiempos de ejecución

Se han medido los tiempos de ejecución de los algoritmos sobre el sistema de MapReduce dejando los siguientes resultados.

[3] www.cloudera.com

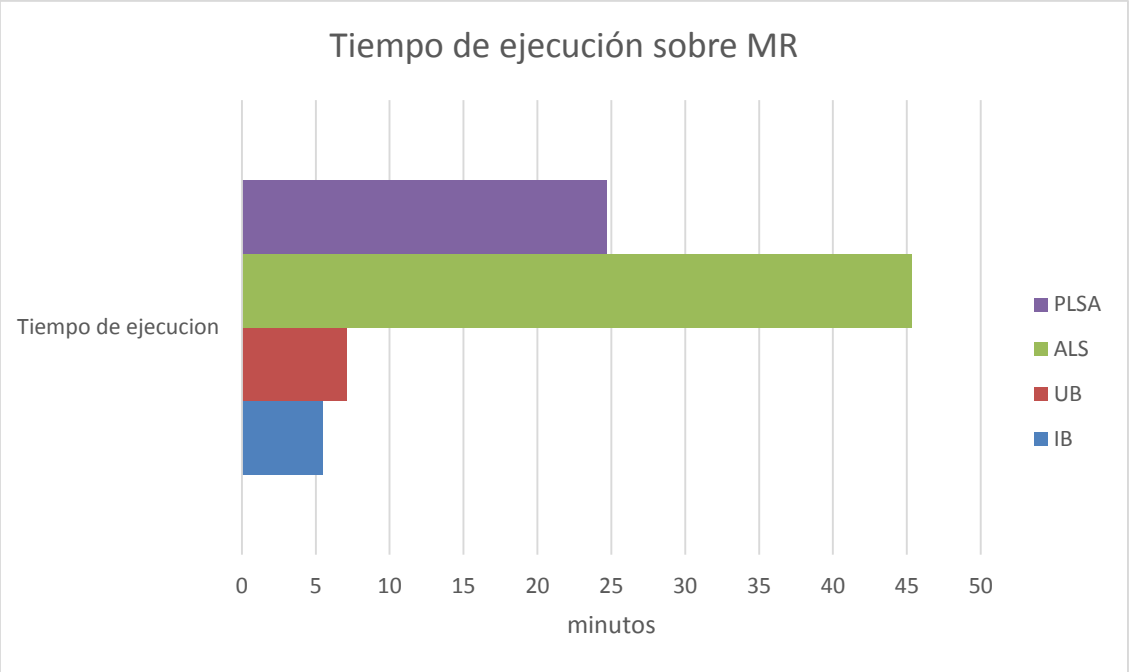



*Figura 1 – Tiempos de ejecución sobre MapReduce*

En la figura 1 se aprecia con enorme claridad que el ALS es el algoritmo que más tiempo de ejecución requiere, necesitando un tiempo mucho mayor que los de la familia KNN: un 831% más; mientras que en relación al pLSA es un 183% mayor. Como se puede ver los algoritmos iterativos se ven penalizados cuando se implementan sobre un modelo MapReduce.

### 5.2.2 Uso de CPU

También se ha medido el uso de CPU máximo del cluster para cada uno de ellos. Ésta vez el que más porcentaje de CPU consume es el ALS, algo lógico y esperado pues sus operaciones con matrices son las más largas y complejas. No es una diferencia tan grande como en términos de tiempos de ejecución pero sí es destacable, puesto que el user-based el ítem-based y el pLSA solo rondan el 20% mientras que el ALS se acerca al 30%.

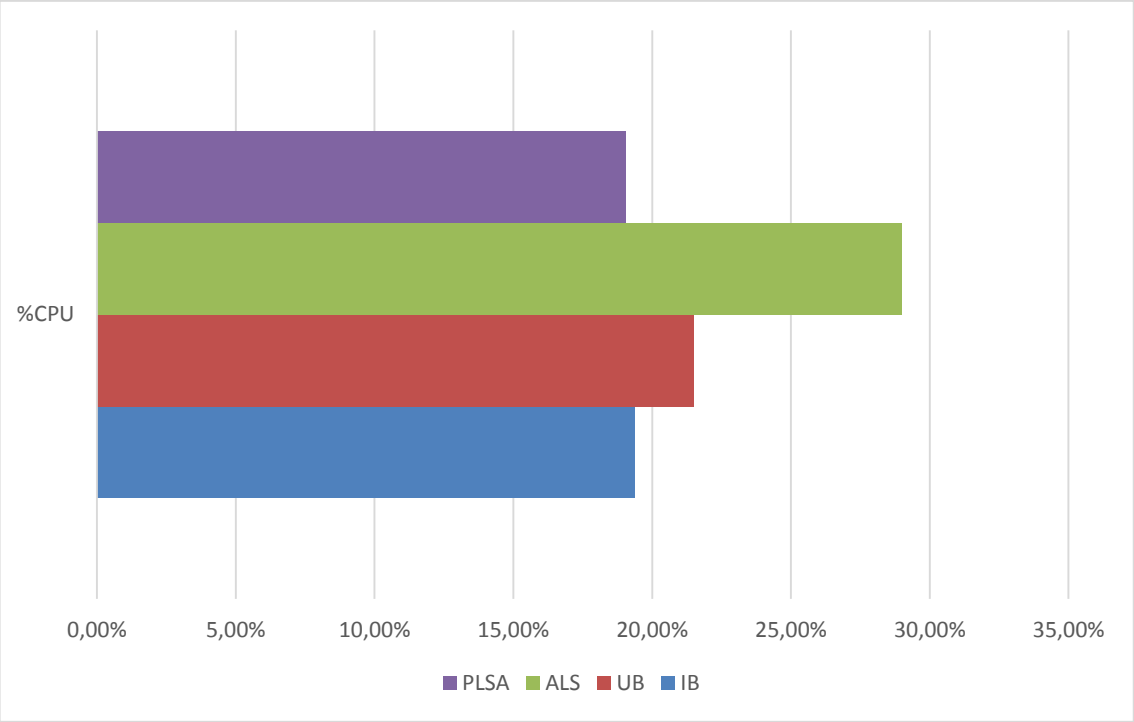



*Figura 2 – Porcentaje de CPU usado sobre MapReduce*

### 5.2.3 I/O de disco y red

En un cluster se trata de minimizar el viaje de datos por red, pues esto además de ser un cuello de botella en cuanto a velocidad puede conllevar un gran coste si los nodos del clúster están distantes. Al ejecutar los cuatro algoritmos sobre el clúster se ha medido el momento máximo de I/O de red y de disco en el cluster.

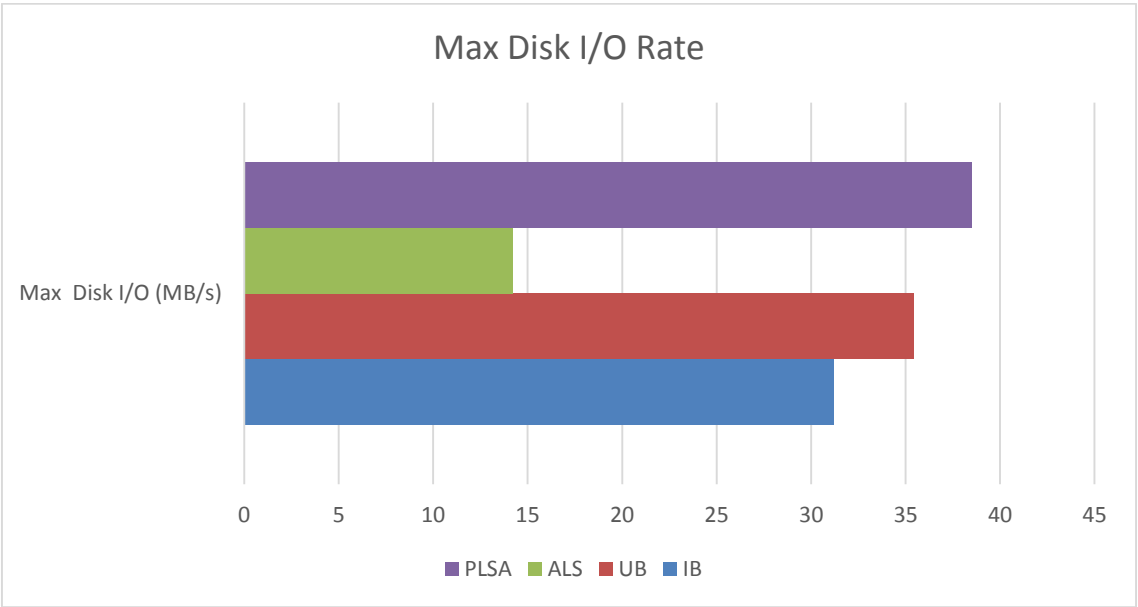



*Figura 3 Tasa máxima I/O de disco sobre MapReduce*

De los resultados observables en la figura 3, donde los más destacable es el poco uso de disco del ALS respecto de los demás, se puede concluir que los algoritmos que solo actúan con fases map reducen significativamente el uso de disco en el cluster. Esto viene

como consecuencia de que al no existir fase reduce, los resultados del map no han de ser procesados por el shuffler para redireccionarlos a los reducers correspondientes. Ésta redirección se lleva a cabo escribiendo en disco en cada nodo correspondiente la porción de archivo resultante a procesar, por lo que al no existir esta fase se reduce significativamente el uso de disco.

Estas teorías son respaldadas por la figura 4, donde las medidas de uso de red van en concordancia a las de uso de disco. De nuevo, el ALS se destaca como eficiente en cuanto a estas medidas mientas que los demás algoritmos, que hacen uso de reducers, ven incrementado el viaje de datos a través de las interfaces de red.

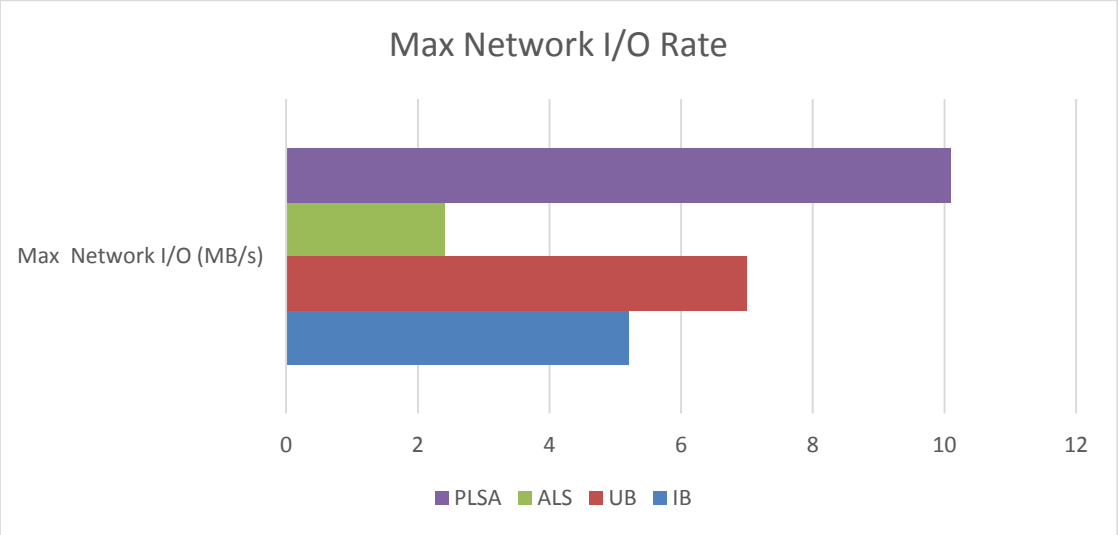



*Figura 4 Tasa máxima de I/O de interfaces de red sobre MapReduce*

### 5.2.4 Uso de tiempo por fase

Otro punto de estudio es cuáles de todas las fases de cada algoritmo requieren más coste computacional. Para ello se ha medido el tiempo de ejecución de las diferentes fases por las que pasa cada algoritmo de recomendación para tratar de detectar dónde se sitúan los procesos que más tiempo y esfuerzo conllevan a la máquina.

En los algoritmos de la familia KNN el coste computacional se lo lleva con ventaja el cálculo de las coocurrencias entre los ítems o usuarios, como se aprecia en la figura 5.

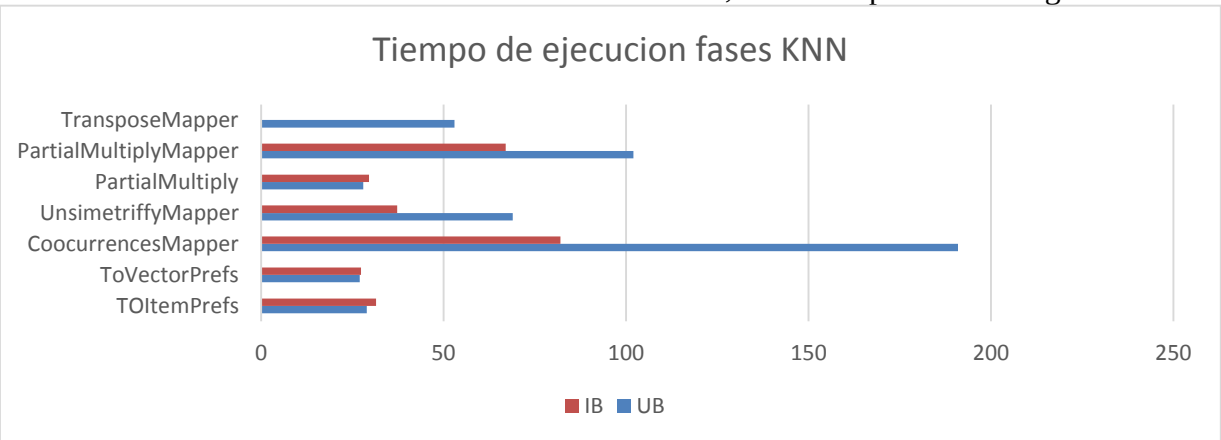



*Figura 5 – Tiempo ejecución fase de KNN sobre MapReduce*

De esta figura se pueden realizar las siguientes conclusiones. El cálculo de las coocurrencias es efectivamente, y como se podría suponer en un principio, lo que más

tiempo consume. Y más especificamente, se ve una diferencia sustancial, aproximadamente el doble, entre la versión del item-based y la del user-based. Esto tiene su explicación en el hecho de que por lo general suele haber una mayor cantidad de usuarios que de items y por lo tanto el número de similitudes a calcular es mayor. En el caso del dataset utilizado esta diferencia es de 6.000 usuarios contra 4.000 peliculas, pero aumenta de manera gradual en mayores datasets (llegando por ejemplo, a 480.000 usuarios y 17.700 películas en el dataset de Netflix).

En el caso del pLSA se ha querido observar cómo se reparte el tiempo una iteración de dicho algoritmo. Para ello se midió el tiempo de ejecución de cada una de sus fases: Expectation y Maximization. Como resultado se obtiene, observable en la figura 6, que el proceso que consume mayor tiempo es el paso de Maximization.


*Figura 6 Tiempo de ejecución fases de pLSA sobre MapReduce*

El algoritmo ALS no presenta ningún dato interesante puesto que sus dos pasos en cada iteración conllevan el mismo tiempo y hacen las mismas operaciones aunque varíen los datos de las matrices P y Q.

## 5.3 Experimentos sobre modelo Pregel

Se ejecutarán las implementaciones de los cuatro algoritmos descritos sobre Giraph. Para todos los experimentos llevados a cabo se utilizarán 6 workers, pues es el número mínimo con el que se ha conseguido correr los experimentos y el cluster no dispone de más espacio para computación de workers.

### 5.3.1 Tiempos de ejecución

Tras analizar los tiempos de ejecución de los algoritmos sobre el modelo de Pregel se puede observar, en la figura 7, un empate técnico entre el pLSA y el User-based con valores muy parecidos, y muy distanciados del ALS y el Item-based, ambos también muy parejos entre sí. El pLSA y el User-Based utilizan aproximadamente un 60% más de tiempo que el Item-Based y el ALS en realizar sus cálculos.

Puede sorprender la gran diferencia entre los dos algoritmos de la misma familia KNN, pero tiene su explicación de nuevo (igual que en el caso de las implementaciones de MapReduce) en que el User-based calcula las similitudes de usuarios en lugar de ítems y el número de usuarios suele ser sustancialmente mayor que el de ítems. Esto, sumado a que las implementaciones sobre Giraph hacen que este tipo de aumentos se note sustancialmente en el número de mensajes enviados, nos obligó a utilizar un parámetro de Giraph denominado OutOfCore. Éste parámetro esta deshabilitado por defecto y abre la posibilidad de guardar en disco particiones del grafo o los mensajes para retomarlos y calcularlos cuando se puedan calcular. En este caso tuvimos que habilitar el OutOfCore para mensajes pues el número de mensajes en el superStep3 era igual a dos veces el número de ratings de usuarios a ítems, conteniendo arrays con los ids de ítems. Esta información era demasiada para el cómputo general por lo que se debió habilitar el OutOfCore para mensajes y procesar los mensajes por partes.

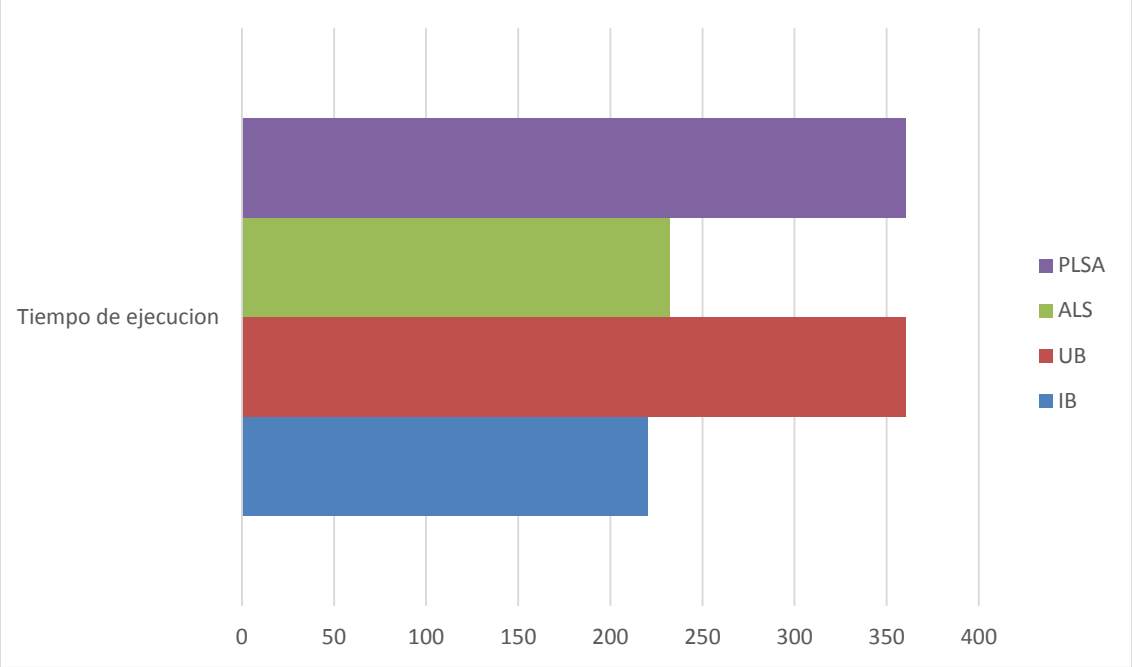



*Figura 7 Tiempos de ejecución sobre modelo Pregel*

### 5.3.2 Uso de CPU

La figura 8 muestra el uso de CPU de los algoritmos sobre Giraph. De nuevo el protagonista vuelve a ser el User-Based por las mismas razones explicadas anteriormente. El simple hecho de tener que estar guardando y recuperando los mensajes ya penaliza su ejecución. Los demás algoritmos se muestran muy parejos en el uso de CPU.

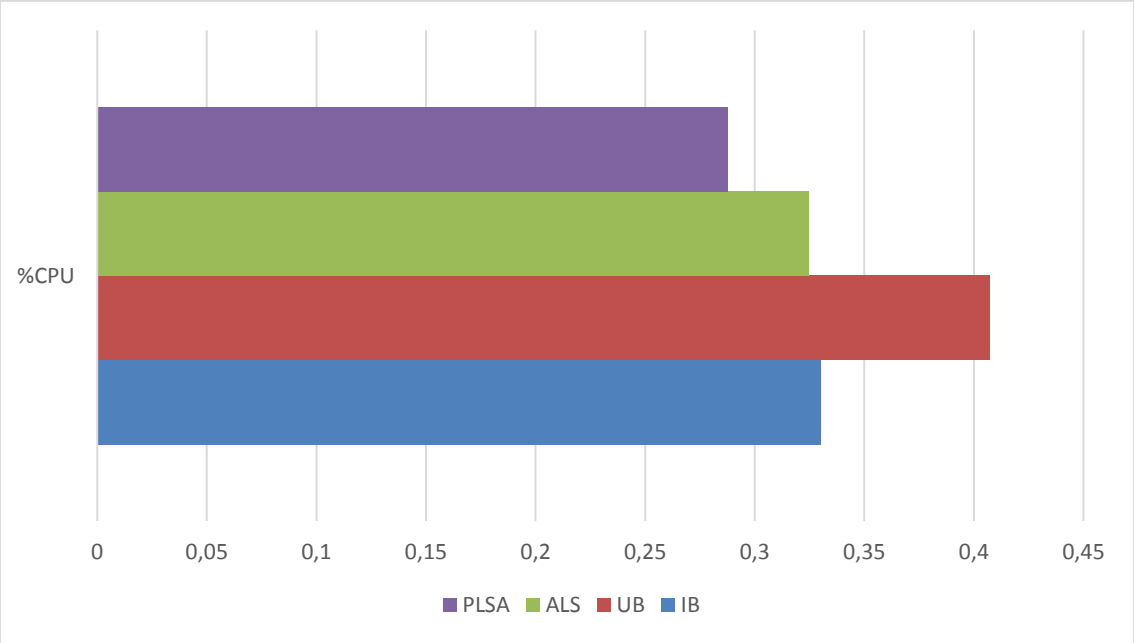



*Figura 8 Porcentaje de CPU máximo sobre Pregel*

### 5.3.3 I/O de disco y red

User-based vuelve a desmarcarse aquí claramente, como se muestra en la figura 9. Mucho que ver en ello tiene la necesidad de habilitar el OutOfCore para guardar los mensajes en disco entre superStep y superStep. Tras esto se puede ver como el pLSA y el ALS utilizan más escritura en disco que el ítem-based, fruto de la necesidad de calcular y guardar los vectores de factores latentes tanto de usuarios como de ítems.

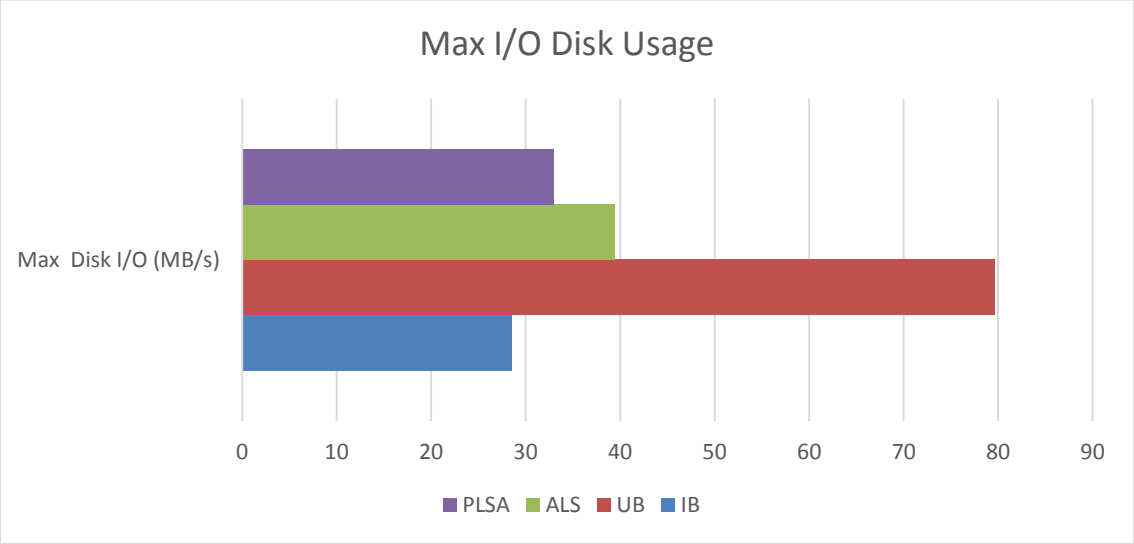



*Figura 9 Tasa máxima de I/O de disco sobre Pregel*

En cuanto a la escritura/lectura en las interfaces de red (figura 10), el algoritmo que más hace uso de este servicio es el pLSA. Esto ocurre a causa de que la información que contienen los mensajes en este algoritmo es mayor que en el resto, pues se envían los vectores de factores latentes a través de ellos.

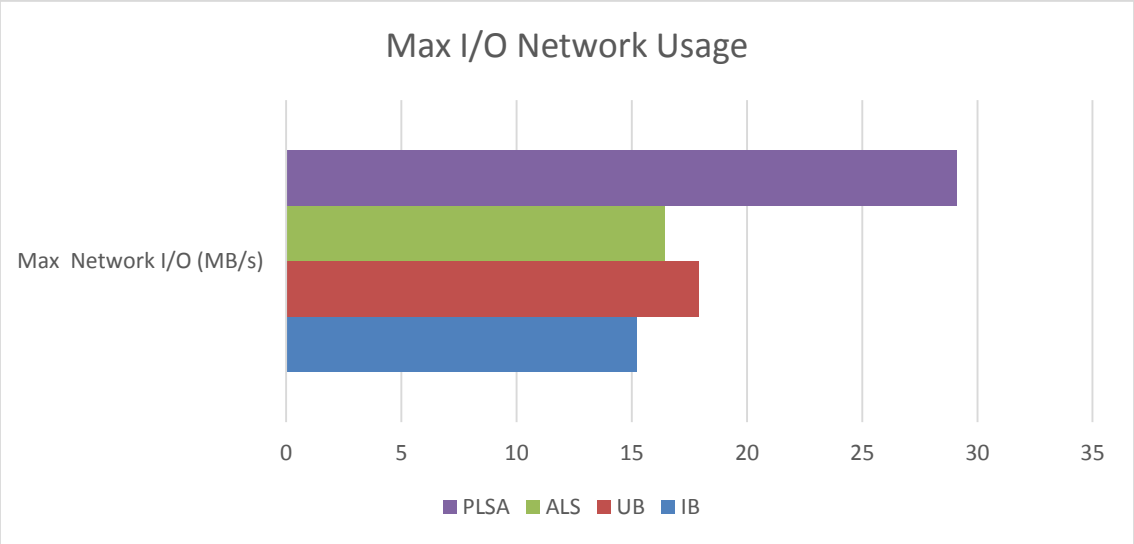



*Figura 10 Tasa máxima de I/O de interfaces de red sobre Pregel*

### 5.3.4 Uso de tiempo por fase

De nuevo es importante observar en qué puntos requiere más tiempo el algoritmo para llevar a cabo sus operaciones.

De nuevo el principal consumidor de tiempo en los algoritmos de KNN es el superStep donde se calculan las similitudes, el número 2 en el caso del User-Based y el 3 en el caso del Item-Based, como se puede ver en la figura 11.

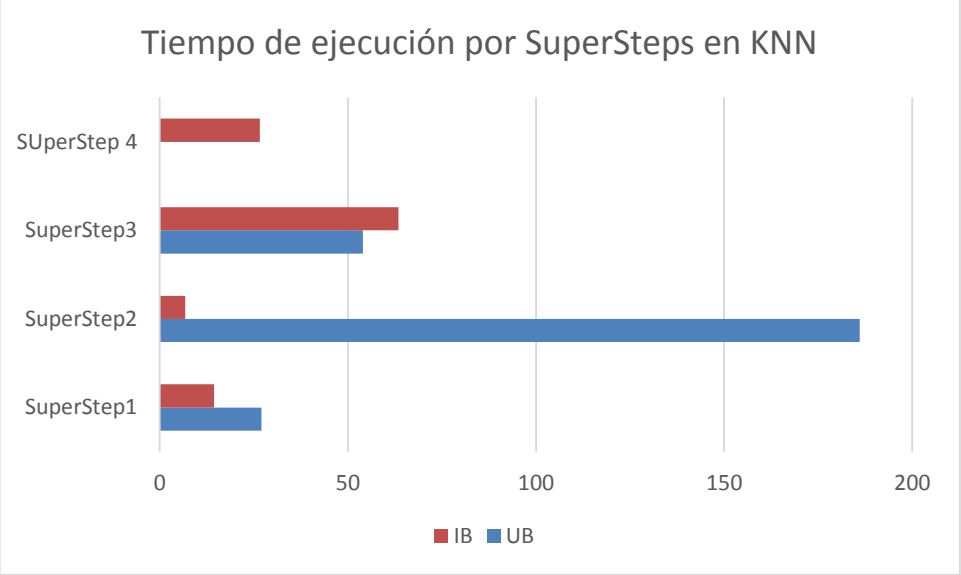



*Figura 11 Tiempo de ejecución de fases en KNN sobre Giraph*

En el caso del pLSA, la fase de expectación se realiza en los superSteps pares, mientras que la maximización se realiza en los impares. Dichos superSteps tienen una media de tiempo de ejecución de 7,7 y 7,9 segundos respectivamente, lo que no supone una gran diferencia.

## 5.4 MapReduce vs Pregel

En la figura 12 se ve con claridad que MapReduce es más lento que Giraph para el cálculo de los algoritmos de recomendación con los que hemos experimentado. Si bien esta tendencia es general en todos ellos, se acentúa notablemente en los algoritmos que conllevan un proceso iterativo, como lo son el ALS y el pLSA. Los procesos iterativos requieren de escritura en disco entre cada fase MapReduce y entre cada iteración, lo que retrasa su ejecución enormemente.

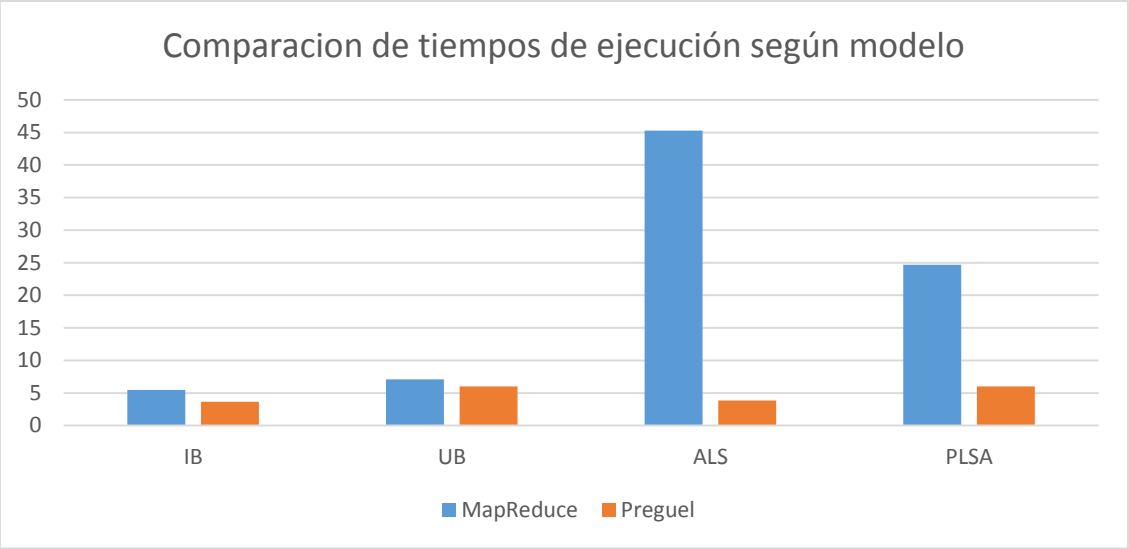



*Figura 12 Comparación de tiempos de ejecución según modelo*

En el aspecto de uso de CPU, el modelo Pregel implementado por Giraph es el que más se desmarca, siendo mayor el porcentaje de CPU usado en todos los algoritmos respecto a sus hermanos de MapReduce, como se puede ver en la figura 13.

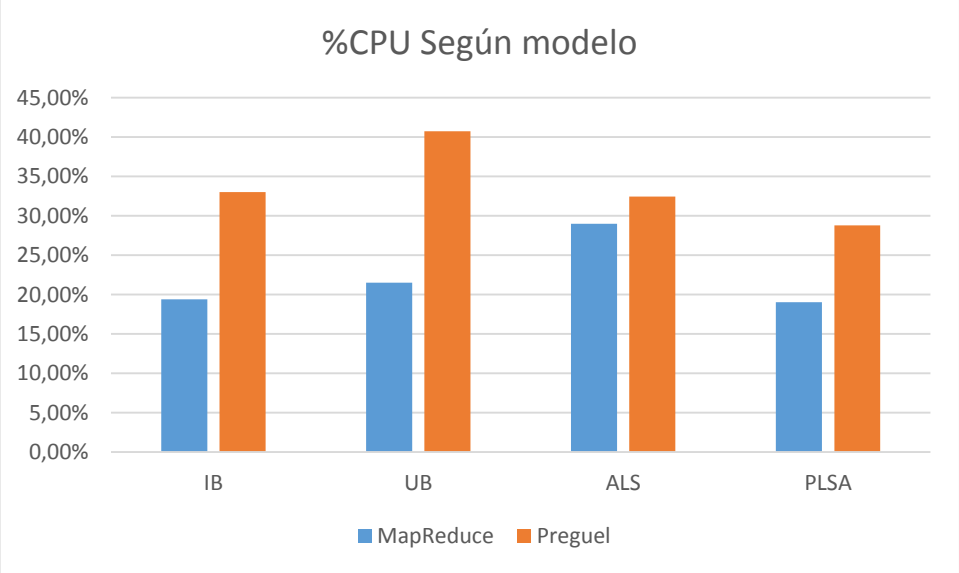



*Figura 13 Comparación de uso de CPU de modelos MapReduce y Pregel*

En cuanto a las tasas de escritura y lectura de disco MapReduce y Pregel están muy parejos en los casos del Item-Based y el pLSA mientras que en ALS y User-Based Pregel requiere mucha una mayor tasa de escritura y lectura (figura 14). Recordemos que el User-Based debió hacer uso del modo OutOfCore de Giraph para poder terminar su ejecución, y que quizá en un cluster más potente no lo hubiera necesitado reduciendo así el uso de disco.

En el ámbito de las tasas de escritura en interfaz de red no hay lugar a discusión: Pregel requiere de un mayor ancho de banda para funcionar. Los algoritmos implementados en Pregel requieren del intercambio de mensajes entre los vértices del grafo, y de estar en diferentes particiones (y por tanto en diferentes nodos de computación del cluster) deberán enviar los mensajes por red, aumentando así la tasa de envío y recibimiento de datos por las interfaces de red, como se aprecia en la figura 15.

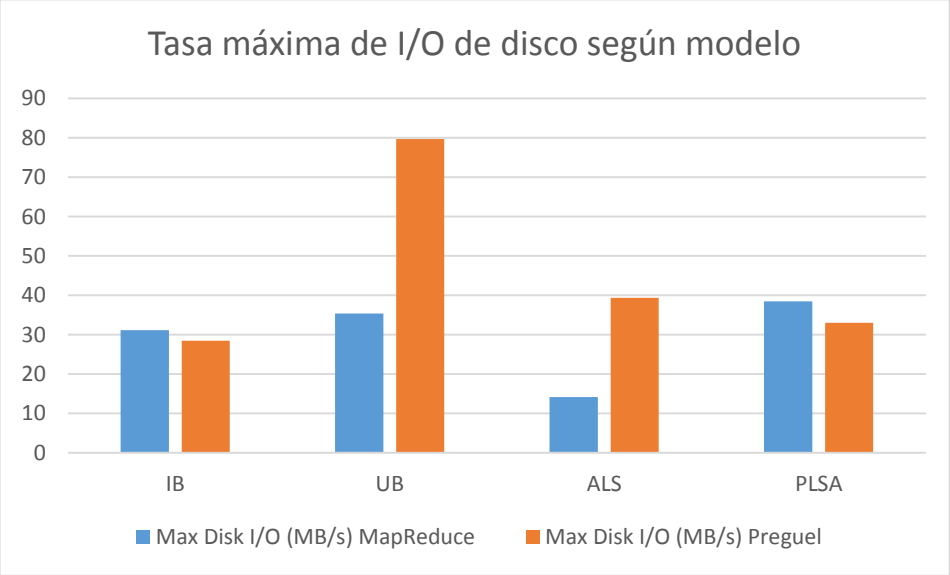



*Figura 14 Comparación de tasa máxima de I/O de disco según modelo*

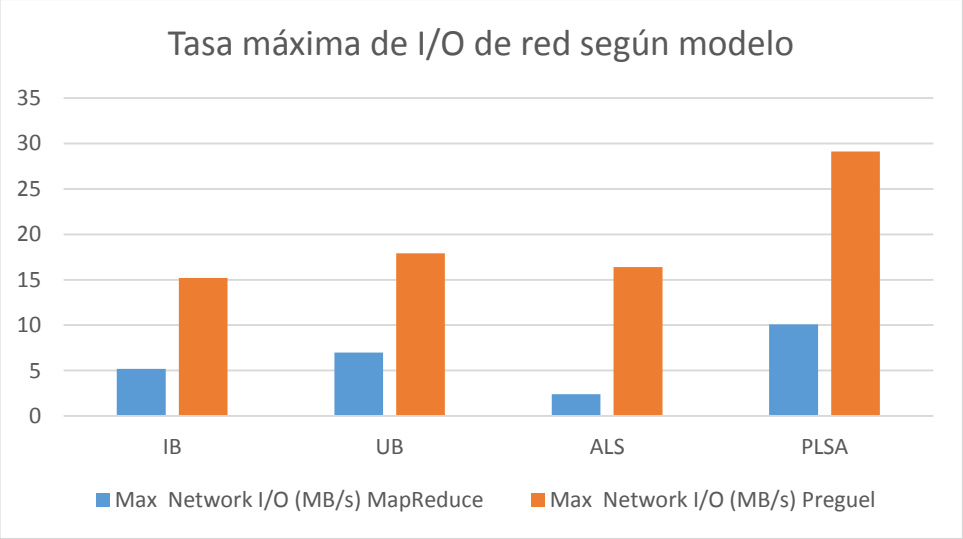



*Figura 15 Comparación de tasa máxima de I/O de red según modelo*

# 6. Conclusiones y Trabajo Futuro

Este trabajo ha brindado la oportunidad de familiarizarse con sistemas de archivos distribuidos como HDFS y con sus modelos de computación paralela existentes hasta el momento como son MapReduce y Pregel. Se ha demostrado la posibilidad de implementar cuatro diferentes sistemas de recomendación sobre ambos modelos. Solo dos de los ocho mostrados en este trabajo habían sido implementados hasta el momento, siendo éstos el ALS y el Item-Based-K-Nearest-Neighbours, implementados por la librería de Mahout sobre el modelo MapReduce.

El modelo Pregel está en una fase muy temprana y se puede decir que las implementaciones sobre él, ya no solo de sistemas de recomendación, brillan por su ausencia. Los cuatro algoritmos de recomendación descritos en este documento han sido implementados desde cero y por primera vez sobre este modelo, demostrando que Pregel puede ser utilizado para realizar sistemas de recomendación, y que muchos problemas pueden ser resueltos si se tratan como un grafo.

Los experimentos sobre ambos modelos han arrojado luz acerca de qué algoritmos funcionan mejor sobre qué modelos, mostrando los puntos fuertes y las debilidades de cada uno. MapReduce consume mucho tiempo de ejecución en la escritura y lectura en disco, y el hecho de tener que adaptar los algoritmos al funcionamiento de Mappers y Reducers puede volverlos contraintuitivos en algunos casos.

Se han comparado las características de ambos modelos frente a frente, llegando a la conclusión de que si bien Pregel es más rápido en sus implementaciones de sistemas de recomendación y mucho más intuitivo en su forma de enfrentar los problemas, consume más ancho de banda, CPU, y disco que MapReduce.

Habiendo demostrado la posibilidad de estas implementaciones, un trabajo futuro podría tratar de comprobar cómo se comportan en términos de escalabilidad y eficiencia estos algoritmos sobre un cluster mayor y con unos datos que no pudieran ser abarcados por una sola máquina. De igual modo, pensar si se podrían optimizar dichas implementaciones, bien reduciendo el número de fases en MapReduce, reduciendo el número de superSteps o información enviada por mensajes en Pregel, o realizar implementaciones sobre nuevos sistemas de computación distribuida residentes en memoria como Spark[4].

[4] www.spark.apache.org

# 7. Referencias

**Aiolli, F.. 2013**: Efficient top-n recommendation for very large scale binary rated datasets. In: ACM Recsys 2013

**Borthakur, D. 2007**. The Hadoop Distributed File System: Architecture and Design.

**Dean, J., Ghemawat, S. 2004**. MapReduce: Simplified Data Processing on Large Clusters.

**Desrosiers, C., Karypis, G**. **2011**: A comprehensive survey of neighborhood-based recommendation methods. In Ricci, F., Rokach, L., Shapira, B., Kantor, P.B., eds.:Recommender Systems Handbook

**Ghemawat, S., Gobioff, H., and Leung, S. 2003**. The Google File System.

**Hilton, G., and Neal, R. 1998.** A view of the EM algorithm that justifies incremental, sparse, and other variants.

**Hofmann, T. 2004**. Latent Semantic Models.

**Hu, Y., Koren, Y. Volinsky, C. 2008.** Collaborative Filtering for Implicit Feedback Datasets.

**Malewicz, G., Austern, M., Bik, A., Dehnert, J, Horn, I.. 2010.** Google Pregel: A System for Large-Scale Graph Processing